RSRE
MEMORANDUM No. 4217
ROYAL SIGNALS & RADAR
ESTABLISHMENT

SPECIFICATION OF VIPER2 IN Z
Author: D H Kemp

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

DTIC
SELECTED
8 MAR 1966

UNLIMITED
ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4217

Specification of Viper2 in Z

Author
D.H. Kemp

Date
October 1988

Summary

As a continuation of the use of the specification language Z which was used to specify the Viper1 microprocessor this paper covers the specification of the Viper2. This was completed before the definitive HOL specification was complete, therefore there is no proof of correspondence between the two. Using Z did highlight inconsistencies in the HOL specification that may not have appeared until later in the specification.

©Controller, HMSO 1988
1. Introduction

This Memorandum is a description of the proposed Viper2 microprocessor using the specification language Z. The description is a continuation of the work done on the Viper1 processor. This is a first attempt to specify the Viper2 and was done in parallel with the specification in Higher Order Logic (HOL). There may therefore be some inconsistencies between this document and the HOL description. Where this occurs the latter should be taken as the definitive description.

In safety critical applications it is necessary to ensure that continued operation or safe shutdown of a system is achieved when erroneous data is input. There are two methods to increase the integrity of a system: to analyse the software for errors and to use a processor that is known to be functionally correct. Further confidence is achieved by using multi-channel systems incorporating processors of dissimilar technologies but with the same functionality. The functionality of any device is determined by the designers' specification. If an error exists in this then all the channels in the system will experience the same common mode error.

By using a number of different methods to specify a processor, errors that may be present in one specification may become apparent in another. This is most effective when the methods used are basically different in character. This can be completed by using proofs of correspondence to confirm that the two texts have the same meaning.

An expertise in the use of Z already exists at RSRE and by using a Z editor and type checker available on the Computing Divisions PerqFlex workstations the task of specifying Viper2 made a useful project for a vacation student, who already had a Knowledge of Z. As a guide to the strategy required for this description J. Bowen's Z specification of the M6800 microprocessor was used.

This report is the first attempt to specify the Viper2 in Z. It makes no attempt to explain the primary constructs of Z, nor to act as a tutorial in the use of Z to specify a microprocessor. Readers not familiar with Z should consult Specification Case studies edited by I. Hayes. Although the specification has been type checked, it has neither been proved equivalent to the HOL specification nor to be free from errors. Any inconsistencies or errors found in this document should be reported back to the Computing Division, RSRE.
2 Basic Functions

2.1 Bits and Words

Bit a (0.1)

Word a ( w:N=Bit | w>0 a dom w = 0 .. (#w - 1) )

Bits are represented as the set of elements with values 0 or 1. Words are represented as a set of partial functions from natural numbers to Bits. The natural numbers correspond to the position of the bit in the word, i.e., the result of \( w(n) \) (the word \( w \) acting on the value \( n \)) gives the \( n+1 \)th bit of the word \( w \).

\[
\begin{align*}
\text{LSB,MSB : Word} & \rightarrow \text{Bit} \\
\text{val : Word} & \rightarrow \mathbb{N} \\
\text{pred : } & \mathbb{N} \\
\text{(set) : (Word, Bit)} & \rightarrow \text{Word} \\
\text{maxval : Word} & \rightarrow \mathbb{N}
\end{align*}
\]

Find the most and least significant bits of the word.

\[
\begin{align*}
\text{val} & : \text{Word} \rightarrow \mathbb{N} \\
\text{pred} & : \mathbb{N} \\
\text{(set)} & : (\text{Word}, \text{Bit}) \rightarrow \text{Word} \\
\text{maxval} & : \text{Word} \rightarrow \mathbb{N}
\end{align*}
\]

\( \text{val} \) returns the natural number represented by the word. Note \( \text{succ}\text{w} \) gives the effect of a right shift, i.e., divide by two, on the word. i.e., if \( \text{succ}\text{w} \) is applied to \( n \) then first \( \text{succ}\text{n} \) is calculated, and then \( \text{w} \) of \( n+1 \) is calculated i.e., the \( n+2 \)th bit is returned rather than the \( n+1 \)th one.

Useful for left shifting (in a similar way to the technique described above):

\[
\begin{align*}
\text{LSB,MSB : Word} & \rightarrow \text{Bit} \\
\text{val : Word} & \rightarrow \mathbb{N} \\
\text{pred : } & \mathbb{N} \\
\text{(set) : (Word, Bit)} & \rightarrow \text{Word} \\
\text{maxval : Word} & \rightarrow \mathbb{N}
\end{align*}
\]

\( \text{pred} \) is defined as \( \text{pred} \text{n} = n - 1 \).

\( \text{(set)} \) is defined as \( \text{(set)} \text{(w, b)} = \text{w} \rightarrow \text{b} \rightarrow (\text{i} \rightarrow \text{b}) \).

The set function returns a word which has all of its bits set to the specified value.

\[
\begin{align*}
\text{LSB,MSB : Word} & \rightarrow \text{Bit} \\
\text{val : Word} & \rightarrow \mathbb{N} \\
\text{pred : } & \mathbb{N} \\
\text{(set) : (Word, Bit)} & \rightarrow \text{Word} \\
\text{maxval : Word} & \rightarrow \mathbb{N}
\end{align*}
\]
\[ (\exists w : \text{Word} \times ((\text{val } w) > \text{maxval } w)) \]

Returns the maximum value which can be stored in the word.

\[
\text{wrd} : N \rightarrow (N \rightarrow \text{Word})
\]

\[
\begin{array}{l}
\forall \text{size} : N ; \forall u : N ; \forall w : \text{Word} =\\
((\text{wrd size valu } = w) =\\
((\text{wrd size valu } = w) \\
(\text{val } w ) = \text{valu mod succ(maxval w))))
\end{array}
\]

The function \text{wrd} returns the word of size \text{size} and set to the value \text{valu} (if that value can be held in a word of that size). Note no algorithm is given for calculating \text{wrd} from its arguments, just the relationships which must hold between the word returned and the input arguments.

\[
(\_\_): (\text{Word} \times \text{Word}) \rightarrow \text{Word}
\]

\[
\forall w1.w2 : \text{Word} =\\
w1.w2 = w1 \cup (\text{pred } w1 \cup w2)
\]

\[
\forall w1.w2 : \text{Word} = w1.w2 = w1 + w2
\]

Concatenate two words together.
2.2 Bitwise functions

\[
\text{not : Bit } \rightarrow \text{ Bit}
\]

\[
\text{not } = (0 \rightarrow 1, 1 \rightarrow 0)
\]

Generate the logical inverse of the input bit.

\[
(_{-}, (_{-}^*), (_{-}^w) : (\text{Bit} \times \text{Bit}) \rightarrow \text{Bit}
\]

\[
(_{-}) = (0.0 \rightarrow 0, 0.1 \rightarrow 1, 1.0 \rightarrow 1, 1.1 \rightarrow 0)
\]

\[
(_{-}^*) = (0.0 \rightarrow 0, 0.1 \rightarrow 1, 1.0 \rightarrow 1, 1.1 \rightarrow 0)
\]

\[
(_{-}^w) = (0.0 \rightarrow 0, 0.1 \rightarrow 0, 1.0 \rightarrow 0, 1.1 \rightarrow 1)
\]

Standard bitwise logical functions. (note o is exclusive or)

Viper\_aux\_1 keeps \text{..}+o, not, word\_maxval, set,pred,word,LSB,MSB, word, Bit
2.3 Logical functions on words

\[ \text{wnot} : \text{Word} \rightarrow \text{Word} \]

\[ \forall w : \text{Word} \]
\[ \text{wnot } w = \text{ Not } w \]

Generate the inverse of the input word.

\[ \text{WordPair} = \{ w : \text{N} \rightarrow (\text{Bit} \times \text{Bit}) \mid w_0 \neq 0 \wedge \text{dom } w = \emptyset \wedge (\text{dom } w) \}
\]

\[ (_\text{pair}_-) : (\text{Word} \times \text{Word}) \rightarrow \text{WordPair} \]

\[ \forall w_1, w_2 : \text{Word} \]
\[ w_1 \text{pair } w_2 = \]
\[ \{ i : \text{N} \mid \exists \text{ dom } w_1 \cap \text{ dom } w_2 \uparrow i = (w_1, w_2) \} \]

Takes a pair of words and represents them as a set of bit pairs, indexed by a single natural number.

\[ (_\text{and}_-), (_\text{or}_-), (_\text{xor}_-) : (\text{Word} \times \text{Word}) \rightarrow \text{Word} \]

\[ \forall w_1, w_2 : \text{Word} \]
\[ w_1 \text{ and } w_2 = \{ (w_1 \text{ pair } w_2) \} \}
\[ w_1 \text{ or } w_2 = \{ (w_1 \text{ pair } w_2) \} \}
\[ w_1 \text{ xor } w_2 = \{ (w_1 \text{ pair } w_2) \} \}

Standard wordwise logical functions.

\[ (_\ll_-) : (\text{Word} \times \text{Bit}) \rightarrow \text{Word} \]

\[ \forall w : \text{Word}; b : \text{Bit} \]
\[ w \ll b = \{ (w_\text{pred } i) \wedge \text{ (pred } i) \} \cup \{\text{bit } i \} \]

\[ (_\gg_-) : (\text{Bit} \times \text{Word}) \rightarrow \text{Word} \]

\[ \forall w : \text{Word}; b : \text{Bit} \]
\[ b \gg w = \{ (w_\text{pred } i) \} \cup \{ \text{bit } i \} \wedge \text{ succ } j \]

Shift right and left while inserting a particular bit into the right or left most position.
2.4 Arithmetic Functions

\[ \text{Value} : \text{Word} \rightarrow \text{Z} \]

\( \text{value} : \text{Word} \rightarrow \text{Z} \)

\[ (\text{MSB} w = 1) \land \text{value} w = \text{val} w - \text{succ} (\text{maxval} w) \lor \\
(\text{MSB} w = 0) \land \text{value} w = \text{val} w \]

Return the integer value represented by the Word. This is using the 2's complement notation. Remember the most significant bit has a weighting of \(-2^{n-1}\). So to cope with negative numbers subtract \(2^n\).

\[ \text{maxpos}, \text{maxneg} : \text{Word} \rightarrow \text{Z} \]

\( \forall w1,w2 : \text{Word} \mid w1 = ((w2)+1) \),
\( \text{maxpos} w1 = \text{maxval} w2 \)
\( \text{maxneg} w1 = (\text{maxval} w2) - (\text{maxval} w1) \)

Return the maximum positive and negative numbers for a word of a particular size.

\[ \text{signextend} : (\text{Word} \times N) \rightarrow \text{Word} \]

\( \forall w1,w2 : \text{Word} \mid \text{length} : N \mid \\
(\text{length} \geq w1) \land (w2 = \text{length}) \implies \\
(\text{w1 signextend length}) = (w2 \text{ set (MSB w1)}) \odot w1 \)

Sign extends the word to the length specified.

\[ \text{pad} : (\text{Word} \times N) \rightarrow \text{Word} \]

\( \forall w1,w2 : \text{Word} \mid \text{length} : N \mid \\
(\text{length} \geq w1) \land (w2 = \text{length}) \implies \\
(\text{w1 pad length}) = (w2 \text{ set 0}) \odot w1 \)

Pad out a word to the new word length with zeros.

\[ \text{trim} : (\text{Word} \times N) \rightarrow \text{Word} \]

\( \forall w : \text{Word} \mid \text{length} : N \mid \text{length} \leq w \implies \\
(\text{w trim length}) = (0 \ldots \text{length}) \odot w \)

Trim a word down to the new word length. Note, use the above with caution, as it simply returns a word with the top b'ts 'trimmed' off. No check is made to ensure that the value of the word has not changed.
\[ (_{\text{plus}}) : (\text{Word} \times \text{Word}) \to \text{Word} \]
\[
\forall \ w_1, w_2, w_3 : \text{Word} \mid (w_1, w_2) \times (w_2, w_3) = (w_2, w_3) \times (w_1, w_2)
\]
\[
(w_2 \text{ plus } w_3 = (w_1 \text{ trim } w_2)) \implies (\text{value } w_1) = (\text{value } w_2) + (\text{value } w_3)
\]

The word returned by plus is the same size as the two input words, and holds the value of the sum of the two words, iff this value can be held in a word of that size.

\[ (_{\text{times}}) : (\text{Word} \times \text{Word}) \to \text{Word} \]
\[
\forall \ w_1, w_2, w_3 : \text{Word} \mid (w_1, w_2) \times (w_2, w_3) = (w_2, w_3) \times (w_1, w_2)
\]
\[
(w_2 \times w_3 = (w_1 \text{ trim } w_2)) \implies (\text{value } w_1) = (\text{value } w_2) \times (\text{value } w_3)
\]

The word returned by times is the same size as the two input words, and holds the value of the product of the two words, iff this value can be held in a word of that size.

\[ (_{\text{minus}}) : (\text{Word} \times \text{Word}) \to \text{Word} \]
\[
\forall \ w_1, w_2, w_3 : \text{Word} \mid (w_1, w_2) \times (w_2, w_3) = (w_2, w_3) \times (w_1, w_2)
\]
\[
(w_2 \text{ minus } w_3 = (w_1 \text{ trim } w_2)) \implies (\text{value } w_1) \leq (\text{value } w_2) - (\text{value } w_3)
\]

The word returned by minus is the same size as the two input words, and holds the value of the difference of the two words, iff this value can be held in a word of that size.

\[ (_{\text{carry}}) : (\text{Word} \times \text{Word}) \to \text{Bit} \]
\[
\forall \ w_1, w_2 : \text{Word} \mid (\text{carry } w_2 = 1) \implies (\text{value } w_1) + (\text{value } w_2) > \text{maxval } w_1
\]

Top level specification of carry, i.e., a carry is generated when the result is larger than the maximum possible value which can be stored.

\[ (_{\text{ncarry}}) : (\text{Word} \times \text{Word}) \to \text{Bit} \]
\[
\forall \ w_1, w_2 : \text{Word} \mid (\text{ncarry } w_2 = 1) \implies (\text{value } w_1) + (\text{value } w_2) > \text{maxval } w_1
\]

Top level specification of carry for multiplication.

\[ (_{\text{borrow}}) : (\text{Word} \times \text{Word}) \to \text{Bit} \]
\[
\forall \ w_1, w_2 : \text{Word} \mid (\text{borrow } w_2 = 1) \implies (\text{value } w_1) < (\text{value } w_2)
\]
Top level specification of Borrow.

```
(_overflow_) : (Word x Word) -> Bit

∀ w1, w2 : Word | w1 = w2 * 
          (w1 overflow w2 = 1) == 
            ( (value w1) * (value w2) < maxpos w1 ) v 
            ( (value w1) * (value w2) < maxneg w2 )
```

Top level specification of overflow, i.e., overflow when the sum is greater than the largest positive value which can be held, or less than the largest negative number.

```
(_overflow_) : (Word x Word) -> Bit

∀ w1, w2 : Word | w1 = w2 * 
          (w1 overflow w2 = 1) == 
            ( (value w1) * (value w2) < maxpos w1 ) v 
            ( (value w1) * (value w2) < maxneg w2 )
```

Top level specification of overflow for multiplication.

```
(_underflow_) : (Word x Word) -> Bit

∀ w1, w2 : Word | w1 = w2 * 
          (w1 underflow w2 = 1) == 
            ( (value w1) - (value w2) < maxpos w1 ) v 
            ( (value w1) - (value w2) < maxneg w2 )
```

Top level specification of overflow on subtraction.

```
(_equal_) : (Word x Word) -> Bit

∀ w1, w2 : Word | w1 = w2 * 
          (w1 equal w2 = 1) == (val w1 = val w2)
```

Returns 1 if the two numbers are the same.

```
(_less_) : (Word x Word) -> Bit

∀ w1, w2 : Word | w1 = w2 * 
          (w1 less w2 = 1) == (value w1 < value w2)
```

Returns 1 if the first number is less than the second.

This completes the underlying theory of representing natural number arithmetic by operations on vectors of bits.
3 Viper Specifics

3.1 Word Lengths

Word₆₄ a Word | w₄ = 64
--- For Double length integers

Word₃₂ a Word | w₄ = 32
--- For Data words

Word₂₀ a Word | w₄ = 20
--- For Address words

Word₈ a Word | w₄ = 8
--- For the function select

Word₆ a Word | w₄ = 6
--- For the destination select

Word₄ a Word | w₄ = 4
--- For the register and memory select

Word₂ a Word | w₄ = 2
--- For the comparison select and flags

Address a Word₂₀
Data a Word₃₂
Flag a Word₁

---

Values

<table>
<thead>
<tr>
<th>one</th>
<th>zero</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

value zero = 0
value one = 1
True = (0-1)
False = (0-0)
3.2 Memory

The definition of the memory and peripheral spaces, and the behaviour of these two regions.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Mem : Address → Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERIspace : Address → Data</td>
<td></td>
</tr>
<tr>
<td>RAMspace : Address → Data</td>
<td></td>
</tr>
<tr>
<td>i0 : Bit</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
(\text{i0} = 0) & \Rightarrow (\text{Mem} = \text{RAMspace}) \\
(\text{i0} = 1) & \Rightarrow (\text{Mem} = \text{PERIspace})
\end{align*}
\]

If i0 is zero then all memory reads are from the RAM space. If i0 is one then all of the reads are from the PERI space.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Mem : Address → Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory'</td>
<td></td>
</tr>
<tr>
<td>δMem : Address → Data</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
(\text{i0} = 0) & \Rightarrow (\text{RAMspace}' = \text{RAMspace} \oplus \delta\text{Mem}) \\
(\text{i0} = 1) & \Rightarrow (\text{RAMspace}' = \text{RAMspace})
\end{align*}
\]

If i0 is zero then any writes will affect the value in the memory. If however i0 is one there are no changes to RAM. Note changes to PERI are not modeled.

<table>
<thead>
<tr>
<th>Memory</th>
<th>δMemory</th>
</tr>
</thead>
<tbody>
<tr>
<td>δMem = 0</td>
<td></td>
</tr>
</tbody>
</table>

No change in memory.
3.3 Registers

The specification of the ViperZ registers.

[RegName]

register a ( r: RegName = word | #>0 )

The registers are the partial function from register names to words.

Reg : RegName = Word

| # n : RegName; Regs = Register
| Reg n = Regs n

Returns the value in the register given as input.

| GeneralPurposeRegisters
| A.X.Y.Z1.Double = RegName
| Reg A e Word32
| Reg X e Word32
| Reg Y e Word32
| Reg Z1 e Word32
| Reg Double e Word64
| Reg Double = (Reg A) ^ (Reg Z1)

The four general purpose read write registers (note X, Y, Z) are index registers. The register double is the concatenation of the A and Z registers.

| AddressRegisters
| F.S.U.P = RegName
| Reg F e Word20
| Reg S e Word20
| Reg U e Word20
| Reg P e Word20

The four addressing registers. The frame pointer F points to the start of the current stack frame. The frame size S is the size of the current stack frame (i.e., the stack frame goes from F to F+S). The stack limit U is the furthest up the stack is allowed to grow. Finally, the program counter P is the position in memory where the current instruction was read from.
OtherRegisters

- D: Watchdog.Temp
  - RegName
  - Reg D: Word32
  - Reg Watchdog: Word32
  - Reg Temp: Word32

The three remaining registers. The D register is the error message register. If an error occurs, then the error code for that particular error is placed in D. The Watchdog register is used when operating in untrusted mode. The value in Watchdog is the number of clock cycles left to complete any untrusted operations. The registers Temp hold the next 32-bit instruction to be executed.

ProcessFlags

- B: Postcall.Trust
  - RegName
  - Reg B: Word1
  - Reg Postcall: Word1
  - Reg Trust: Word1

The three process flags are held as one bit registers. The B flag contains the result from various comparisons or unsigned arithmetic. The Postcall flag is there to ensure that the Enter instruction always occurs after a call instruction, and never anywhere else. It is set true after a call and cleared during an enter. The Trust flag determines whether the machine is in trusted or untrusted mode.

ErrorFlags

- E, IA, IX, IY, IZ, IB, WE, NoStack, NoSize, NoLimit
  - RegName
  - Reg E: Word1
  - Reg IA: Word1
  - Reg IX: Word1
  - Reg IY: Word1
  - Reg IZ: Word1
  - Reg IB: Word1
  - Reg WE: Word1
  - Reg NoStack: Word1
  - Reg NoSize: Word1
  - Reg NoLimit: Word1

The error flags. The E flag is set true if there has been an error. This is utilised by the Jump on error and Call on error instructions. The IA, IX, IY, IZ, and IB flags show whether a register holds an invalid value, i.e., IA is true if A has not been loaded since the machine started, or since an error occurred. The WE register is set if the Watchdog timer has expired (hence WE). This flag will cause an
error to occur if it is set while the machine is in untrusted mode. It is ignored in trusted mode. The NoStack, NoSize and NoLimit Flags are set true if the F, S and U registers have not been set.

```
Regs & ErrorFlags A
ProcessFlags A
OtherRegisters A
AddressRegisters A
GeneralPurposeRegisters
```

The Viper2 register types

```
<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regs</td>
</tr>
<tr>
<td>Registers : Register</td>
</tr>
</tbody>
</table>
```

The registers at split time consist of a 'bank' of registers and the Viper2 register types.

```
<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
</tr>
<tr>
<td>registers'</td>
</tr>
<tr>
<td>newp : Address</td>
</tr>
<tr>
<td>NewWatchdog : Data</td>
</tr>
<tr>
<td>NewIE : Flag</td>
</tr>
<tr>
<td>δReg : RegName = Word</td>
</tr>
<tr>
<td>Registers' = Registers o (P=newp,IE=NewIE)</td>
</tr>
<tr>
<td>o (Watchdog=NewWatchdog) o δReg</td>
</tr>
</tbody>
</table>
```

The Viper2 registers: The new values of the registers are the same as the old value, apart from the three registers which are always updated (the program counter watchdog timer and watchdog expired flag). These can be overwritten by any modifications to them in δReg. Any other changes in the registers (due to the various instructions) are also contained in δReg.

```
<table>
<thead>
<tr>
<th>SRegisters</th>
</tr>
</thead>
<tbody>
<tr>
<td>δRegisters</td>
</tr>
<tr>
<td>δReg = {}</td>
</tr>
</tbody>
</table>
```

All of the registers remain the same (apart from the three above)
3.4 Clock

The ViperZ clock is not represented in the HOL specification. A definition is included here for completeness.

\[
\begin{array}{l}
\text{Clock} \\
\text{Clk : N}
\end{array}
\]

Clock simply counts up from 0.

\[
\begin{array}{l}
\text{AClock} \\
\text{Clock} \\
\text{Clock'} \\
\text{Cycles : N} \\
\text{Clk'} = \text{Clk} + \text{Cycles}
\end{array}
\]

Cycles is the number of cycles needed to complete the present instruction. The parameter cycles is used by the Watchdog timer.
3.5 Stop

The definition of the Stop Flag:

\[
\text{Stop} \\
\quad \text{stop : Bit}
\]

The single bit to determine whether the machine is stopped or not:

\[
\text{Stop} \\
\quad \text{ARegisters} \\
\quad \text{Stop} \\
\quad \text{Stop'} \\
\quad \text{Values} \\
\quad \text{val : Bit}
\]

\[
\text{stop } = 0 \\
\text{newp } = \text{Reg (P) plus one}
\]

The machine has not stopped. The new value of the Program Counter is P+1. The value of stop' is set later in the specification.
3.6 Viper State

The Viper2 changing state. The change in the Viper state is the change in memory and the change in registers and the change in stop.

```
ArithmeticAndLogicalUnit
  r,m : Data
  offs : Data
  base : Data
  Result : Data
```

The inputs and outputs to/from the ALU. r and m are the two inputs to the ALU and Result is the result from it. Base is the base address to read the memory (m) input to the ALU from and offs is the actual address of the read.
3.7 ViperZ Operation Codes

The ViperZ Op code. Op is the op code and is loaded from the address pointed to by the Program Counter. The op code is the concatenation of the five fields shown: s2, s1, fq, fc and addr. The fc and s2 fields are further subdivided into two 2 bit fields.

The s2 field selects the addressing mode for the m input to the ALU if the instruction is a data operation, or whether the operation is a control or write instruction. The s1 field selects the register (r) input to the ALU or the type of certain load instructions. The fq (functional qualifier) field selects the destination register of the data instructions, or whether the instruction is a control or a write instruction. It also determines the type of call or branch performed (ie absolute or Program Counter relative). The fc (function code) determines the instruction to be performed. Finally the addr field determines the location to jump to, write to, read from etc.
3.8 ViperZ Overall State

Viper2Inputs

attention : Bit
reset : Bit

The two external input lines. These are assumed to be synchronous
lines clocked in at the start of each instruction. The attention line
is set by external devices to inform the processor when they require
attention. It is polled by the Jump and Call on attention
instructions.

ViperZ

∆State
ArithmeticAndLogicalUnit
Viper2OpCode
Viper2Inputs

Values

op = Mem (Reg (P))
reset = 0
(Reg WE = False) = (NewWatchdog=Reg Watchdog)
minus (wrd 32 Cycles))
(NewWE = True) == (Reg Watchdog) borrow (wrd 32 Cycles) = 1)

The op code is the value in the memory location pointed to by
Program Counter. The reset line must be low, otherwise the machine
will reset. The new value of the Watchdog Expired flag will be set to
True if the watchdog counter will become less than zero in the course
of the present instruction (not quite true as WE in fact goes true
immediately the Watchdog timer goes below zero). The watchdog timer is
decremented if the WE flag is not set.

Stopped

Memory
Registers
Stop
Stop'
∆clock

stop = 1
stop' = 1
newp = Reg (P)

The machine has stopped, and cannot restart until there is a Reset.
Viper state unchanged (except P. Watchdog and WE updated)

<table>
<thead>
<tr>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMemory</td>
</tr>
<tr>
<td>URegisters</td>
</tr>
<tr>
<td>UClock</td>
</tr>
<tr>
<td>UStop</td>
</tr>
<tr>
<td>Viper2Inputs</td>
</tr>
<tr>
<td>Values</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{stop} &= 0 \\
\text{reset} &= 1 \\
\text{val newp} &= 0 \\
\text{Reg} &= \text{EE-False, IA-True, IX-True, NoStack-True, NoSize-True,}
\text{ IY-True, IB-True, Trust-True, NoLimit-True, WE-False, Watchdog-((Reg Watchdog) set 1))}
\end{align*}
\]

Machine status on a Reset. All of the Register Illegal flags are set to true (as the registers have not had any values loaded into them yet). The error flag is set false, as is the Watchdog Expired flag. The program counter is set to zero.

Viper2INIT

<table>
<thead>
<tr>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk' = 0</td>
</tr>
</tbody>
</table>

Machine on start up.

Viper2_machine_state keeps

\[
\text{&Clock, Stop, UStop, &Viper2, &Viper2, &Viper2INIT,}
\text{EMemory, EMemory, registers, URegisters, UClock,}
\text{Address, Data, Flag, Memory, Reset, Reg, Word,}
\text{Word2, Word3, Word4, Word5, Word6, &Word, Stopped,}
\text{RegName}
\]

This section specifies the inputs to the Viper2 ALU.

\[
\text{invalid : Word = Bit}
\]

\[
\text{\text{invalid} W = 1 \Rightarrow (val w > maxval (word 20 0))}
\]

Returns True if the value is greater than can be held in a 28-bit word.
The instruction being executed is illegal. The error code of the particular error is returned in ErrorValue.

The program counter is about to carry, and the current instruction is not a jump. (there is no need to cause an error if the instruction is a jump, as there is no 'return' as in a call instruction).
4 Viper Operations

4.1 ViperZ ALU

This section specifies the vi input to the ViperZ ALU.

<table>
<thead>
<tr>
<th>RegisterSelect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔViper2</td>
</tr>
<tr>
<td>(val s1 = 0) ← (r = Reg A)</td>
</tr>
<tr>
<td>(val s1 = 1) ← (r = Reg X)</td>
</tr>
<tr>
<td>(val s1 = 2) ← (r = Reg Y)</td>
</tr>
<tr>
<td>(val s1 = 3) ← (r = Reg Z)</td>
</tr>
</tbody>
</table>

Select the register to be used as the r input to the ALU.

<table>
<thead>
<tr>
<th>DataInstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔViper2</td>
</tr>
<tr>
<td>val s2 = 15</td>
</tr>
</tbody>
</table>

The instruction is a data instruction. If s2 was 15 then it would be a control or write instruction.
### 4.2 Addressing Modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Data/Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GlobalAddressing</strong></td>
<td>[val s2u = 0&lt;br&gt;Reg Trust = True&lt;br&gt;base = addr pad 32]</td>
</tr>
<tr>
<td><strong>StackRelativeAddressing</strong></td>
<td>[val s2u = 1&lt;br&gt;base = (addr pad 32) plus (Reg F pad 32)]</td>
</tr>
<tr>
<td><strong>ProgramCounterRelativeAddressing</strong></td>
<td>[val s2u = 2&lt;br&gt;base = (addr pad 32) plus (Reg P pad 32)]</td>
</tr>
</tbody>
</table>

**Relative addressing mode.** The base address is the address in the Op code. The machine must be in trusted mode.

**Stack relative addressing.** This gives access to local routine variables. The base address is the frame pointer offset by the address from the Op code. No check is made here to see if the address calculated is in the current stack frame. This is done in a later error frame.

**Program Counter relative addressing.** This gives access to constants embedded in the program. This allows routines to be relocatable in memory (ie standard ROMs can be bought which can plug straight into a system). The base address is the program counter plus the input address.

---

**AddressBases & GlobalAddressing + StackRelativeAddressing + ProgramCounterRelativeAddressing**

The three basic addressing modes. The base address is offset by the various index registers (or not in the case of absolute addressing).

<table>
<thead>
<tr>
<th>AddressBases &amp; GlobalAddressing + StackRelativeAddressing + ProgramCounterRelativeAddressing</th>
<th>Data/Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AbsoluteAddressing</strong></td>
<td>[val sZu = 0&lt;br&gt;offs = base]</td>
</tr>
</tbody>
</table>

**Absolute Addressing.** The location to read from is simply the base address defined above.
Indexed Addressing using the X index register. The location to read from is the base address plus the value contained in the X index register. Note the value in X can be either a positive or a negative value. This can be used to index arrays etc.

Indexed Addressing using the Y index register. The location to read from is the base address plus the value contained in the Y index register. Note the value in Y can be either a positive or a negative value. This can be used to index arrays etc.

Indexed Addressing using the Z index register. The location to read from is the base address plus the value contained in the Z index register. Note the value in Z can be either a positive or a negative value. This can be used to index arrays etc.

Indexed Addressing using the Z index register. The location to read from is the base address plus the value contained in the Z index register. Note the value in Z can be either a positive or a negative value. This can be used to index arrays etc.

All of the simple addressing modes.

The simple addressing modes. This does not include the case of the monadic instructions, where a memory read will not be taking place, or the Immediate and Register addressing modes, where no memory read is taking place. The value on the io pin is zero, so the word read in is read from the RAM space. The input to the ALU is the value in the location pointed to by offset. The case of offset being outside the 20 bit address space is dealt with in the errors later.
Immediate Addressing

Data Instruction

(val $a_2 = 12)_a (m = addr pad 32) +
(val $a_2 = 13)_a (m = \text{wnot} (addr \text{ pad } 32))

The two Immediate Addressing modes. The $m$ input to the ALU is the
value in the address field padded with zeros to 32 bits, if $a_2$ is 12.
If $a_2$ is 13 then the $m$ input is this value inverted (i.e., 1's
complement). This allows both negative and positive values to be used
as constants.
4.3 Access to General Purpose Registers

In this case the \( m \) input to the ALU is one of the general purpose registers \( A, X, Y, \) or \( Z \). Which register is used is determined by the bottom two bits in the address field of the op code.

\[
\begin{align*}
\text{UseRegisterA} & \quad \text{RegisterAddress} \\
(\text{val addr}) \mod 4 &= 0 \\
\quad m &= \text{Reg A}
\end{align*}
\]

The \( A \) register is used as the \( m \) input to the ALU.

\[
\begin{align*}
\text{UseRegisterX} & \quad \text{RegisterAddress} \\
(\text{val addr}) \mod 4 &= 1 \\
\quad m &= \text{Reg X}
\end{align*}
\]

The \( X \) register is used as the \( m \) input to the ALU.

\[
\begin{align*}
\text{UseRegisterY} & \quad \text{RegisterAddress} \\
(\text{val addr}) \mod 4 &= 2 \\
\quad m &= \text{Reg Y}
\end{align*}
\]

The \( Y \) register is used as the \( m \) input to the ALU.

\[
\begin{align*}
\text{UseRegisterZ} & \quad \text{RegisterAddress} \\
(\text{val addr}) \mod 4 &= 3 \\
\quad m &= \text{Reg Z}
\end{align*}
\]

The \( Z \) register is used as the \( m \) input to the ALU.

RegisterAddressing \& UseRegisterA \& UseRegisterX \& UseRegisterY \& UseRegisterZ

The four cases of register addressing.

MemoryRead \& ImmediateAddressing \& IndexAddressing \& RegisterAddressing

The fifteen cases of memory addressing for the fifteen values \( s2 \) can have for any data instruction.
4.4 Illegal Addressing Operations

StackNotSet

DataInstruction
ErrorInstruction

val sZu = 1
Reg NoStack = True
Mem = ()

Stack Relative addressing has been specified, however no stack has been set up (i.e. no value has been loaded into F).

UnsetX

AddressBases
ErrorInstruction

val s21 = 1
Reg IX = True
Mem = ()

The X register has been selected as the index register to be used, but it has either not been loaded, or an error has occurred in untrusted mode and all of the registers have been marked illegal.

UnsetY

AddressBases
ErrorInstruction

val s21 = 2
Reg IY = True
Mem = ()

The Y register has been selected as the index register to be used, but it has either not been loaded, or an error has occurred in untrusted mode and all of the registers have been marked illegal.

UnsetZ

AddressBases
ErrorInstruction

val s21 = 3
Reg IZ = True
Mem = ()

The Z register has been selected as the index register to be used, but it has either not been loaded, or an error has occurred in untrusted mode and all of the registers have been marked illegal.

UnsetAddressingRegister a StackNotSet
v UnsetZ
v UnsetY
v UnsetX
The four cases of illegally used registers.

**IllegalStackAddress**
- IndexedAddressing
- ErrorInstruction

- `val fc = 13`
- `val sz = 1`
- `((val offs) < (val (Reg F)) or (val offs) > (val (Reg F)) + (val (Reg S)))`
- `6Mem = {}`

**IllegalReadAddress**
- IndexedAddressing
- ErrorInstruction

- `val fc = 13`
- `val sz = 12`
- `invalid offs = 1`
- `6Mem = {}`

This is the only check that is needed to see if the address is valid. This is because the base address is at most a 21 bit number, so there can be no overflow on the first addition. The index register added to this base value can be one of four cases:

1. The index register holds a +ve number and the result causes overflow. Then the MSB of result is one and hence above predicate detects the invalid address.

2. The index register holds a +ve number and no overflow occurs. Then the address is valid iff the above predicate holds.

3. The index register holds a -ve number and a carry occurs. Then the result must be a positive number less than the base. It is valid.

4. The index register is negative and no carry occurs, ie the index register held a negative number which was 'larger' than the base. This is detected as for -ve numbers MSB = 1, and hence invalid address.

The case of -ve index register and overflow cannot occur as base is ALWAYS positive.
4.5 Illegal Source Registers

RegisterAInvalid
RegisterAddress
ErrorInstruction
(val addr) mod 4 = 0
Reg IA = True
Mem = ()

Register addressing has been specified, with the m input to the ALU coming from the A register. This register however does not contain valid data.

RegisterXInvalid
RegisterAddress
ErrorInstruction
(val addr) mod 4 = 1
Reg IX = True
Mem = ()

Register addressing has been specified, with the m input to the ALU coming from the X register. This register however does not contain valid data.

RegisterYInvalid
RegisterAddress
ErrorInstruction
(val addr) mod 4 = 2
Reg IY = True
Mem = ()

Register addressing has been specified, with the m input to the ALU coming from the Y register. This register however does not contain valid data.

RegisterZInvalid
RegisterAddress
ErrorInstruction
(val addr) mod 4 = 3
Reg IZ = True
Mem = ()

Register addressing has been specified, with the m input to the ALU coming from the Z register. This register however does not contain valid data.

RegisterInvalid & RegisterAInvalid & RegisterXInvalid & RegisterYInvalid & RegisterZInvalid
The four cases where an illegal register has been selected to be the input to the ALU.

RegisterSelectUnvalidError

\[ \text{ErrorInstruction} \]

\[ \begin{align*}
( \text{val } s2 & = 15 \land \text{val } fc & = 13 ) \lor \\
( \text{val } fc & = 3 \land \text{val } fc & < 12 ) \lor \\
( \text{val } fc & = 6 )
\end{align*} \]

Error \( \Delta \text{Men} = \{ \} \)

The instruction selected requires a register be the input to the ALU, i.e. either a dyadic data instruction \( s2 \neq 15 \) and \( fc \neq 13 \), a write instruction \( s2 = 15 \), \( fc = 3 \) and \( fc < 12 \) (this last condition because \( fc = 12 \) would give a different error code), or the instruction is decrement with branch on zero.

RegisterSelectAUnvalid

\[ \begin{align*}
\text{val } s1 & = 0 \\
\text{Reg IA} & = \text{True}
\end{align*} \]

The instruction requires the input to the ALU to be the A register but this register does not contain valid data.

RegisterSelectXUnvalid

\[ \begin{align*}
\text{val } s1 & = 1 \\
\text{Reg IX} & = \text{True}
\end{align*} \]

The instruction requires the input to the ALU to be the X register but this register does not contain valid data.

RegisterSelectYUnvalid

\[ \begin{align*}
\text{val } s1 & = 2 \\
\text{Reg IY} & = \text{True}
\end{align*} \]

The instruction requires the input to the ALU to be the Y register but this register does not contain valid data.

RegisterSelectZUnvalid

\[ \begin{align*}
\text{val } s1 & = 3 \\
\text{Reg IZ} & = \text{True}
\end{align*} \]

The instruction requires the input to the ALU to be the Z register but this register does not contain valid data.
The four cases of illegal register being used for the input to the ALU.
4.6 Comparison Operations

Framing schema for comparison operations. All registers are unchanged except for the Program counter. \( B \) is set in the various comparisons below.

GreaterThanOrEqualTo

\[
\text{CompareFrame} \\
\text{Register-Select} \\
\text{Memory-Read} \\
\text{Bresult} = \text{Word}_1 \\
\delta \text{Mem} = ()
\]

\[
\text{val fc} = 0 \\
\text{Bresult} = \text{wrd} 1 \ (\text{not} \ (r \text{ less} \ m))
\]

Bresult is set true if the \( r \) input is greater than or equal to the \( m \) input.

EqualTo

\[
\text{CompareFrame} \\
\text{val fc} = 1 \\
\text{Bresult} = \text{wrd} 1 \ (r \text{ equal} \ m)
\]

Bresult is set true if the \( r \) input is equal to the \( m \) input.

GreaterThan

\[
\text{CompareFrame} \\
\text{val fc} = 2 \\
\text{Bresult} = \text{wrd} 1 \ (\text{not}((r \text{ less} \ m) + (r \text{ equal} \ m)))
\]

Bresult is set true if the \( r \) input is greater than the \( m \) input.

UnsignedLessThan

\[
\text{CompareFrame} \\
\text{val fc} = 3 \\
\text{Bresult} = \text{wrd} 1 \ (r \text{ borrow} \ m)
\]

Bresult is set true if the \( r \) input, treated as an unsigned integer, is less than the \( m \) input.
\[
\text{AndEqualZero}
\]
\[
\text{CompareFrame}
\]
\[
\text{val } f_c = 4
\]
\[
\text{Bresult } = \text{and } 1 \text{ ((r and m) equal (zero))}
\]

Bresult is set true if the r input logically anded with the m input is equal to zero.

\[
\text{CompOp} = \text{AndEqualZero} \lor \text{UnsignedLessThan} \lor \text{GreaterThan}
\]
\[
\lor \text{EqualTo} \lor \text{GreaterThanOrEqualTo}
\]

The five basic comparison operations. B is loaded with the following

\[
\text{Condition}
\]
\[
\text{CompOp}
\]
\[
\text{val } f_q = 0
\]
\[
\text{RReg } = \{\text{B-Bresult}, \text{IB-False}\}
\]

B is loaded with Bresult. The Illegal B flag is set false to show that the B register contains valid information.

\[
\text{NotCondition}
\]
\[
\text{CompOp}
\]
\[
\text{val } f_q = 1
\]
\[
\text{RReg } = \{\text{B-not(Bresult)}, \text{IB-False}\}
\]

B is loaded with not Bresult. The Illegal B flag is set false to show that the B register contains valid information.

\[
\text{BorCondition}
\]
\[
\text{CompOp}
\]
\[
\text{val } f_q = 2
\]
\[
\text{RReg } = \{\text{B-or Reg(B) or Bresult}, \text{IB-False}\}
\]

B is loaded with Bresult or B. The Illegal B flag is set false to show that the B register contains valid information.

\[
\text{BorNotCondition}
\]
\[
\text{CompOp}
\]
\[
\text{val } f_q = 3
\]
\[
\text{RReg } = \{\text{B-or Reg(B) or not(Bresult)}, \text{IB-False}\}
\]

B is loaded with not Bresult or B. The Illegal B flag is set false to show that the B register contains valid information.

Compare a Condition \lor BorNotCondition
\lor BorCondition \lor NotCondition
The four operations loading B with a result. There are $15 \times 4 \times 20 = 1200$ compare operations out of the possible $2^{12}$ Viper2 operations.
4.7 Viper2 Arithmetic

Framing schema for all of the ALU operations. Note memory cannot be changed. \( \overline{BB} \) holds any changes to the B register.

SignedAdd

\[
\begin{align*}
\text{ALUInstruction} & \\
\text{RegisterSelect} & \\
\text{MemoryRead} & \\
\overline{BB} : \text{RegName} & \to \text{Word} \\
\text{Mem} & = ()
\end{align*}
\]

Add \( r \) to \( m \). There is no check for overflow; this is done later in an error schema.

UnsignedAdd

\[
\begin{align*}
\text{ALUInstruction} & \\
\text{val fc} & = 6 \\
\text{Result} & = r \text{ plus } m \\
\overline{BB} & = ()
\end{align*}
\]

Add \( r \) to \( m \), setting \( B \) if there is a Carry. \( IB \) is set false whatever the result.

SignedSubtract

\[
\begin{align*}
\text{ALUInstruction} & \\
\text{val fc} & = 7 \\
\text{Result} & = r \text{ minus } m \\
\overline{BB} & = ()
\end{align*}
\]

Subtract \( r \) from \( m \). There is no check for underflow; this is done later in an error schema.
UnsignSubtract

ALUInstruction

val fc = 0
Result = r minus m
\( \Delta B = \text{B-wrd 1 (r borrow m).IB-False} \)

Subtract m from r, and setting B if there is a Borrow. IB is set false whatever the result.

SignedMultiply

ALUInstruction

val fc = 12
Result = r times m
\( \Delta B = \{ \} \)

Multiply r by m. There is no check for overflow, this is done later in an error schema.

ArithmeticOp = (UnsignedAdd v SignedSubtract v SignedMultiply v UnsignedSubtract v SignedAdd)

The five arithmetic operations. There are 15 * 4 * 5 = 300 possible operations (15 addressing modes by four register inputs by five possible operations).
4.8 Logical Operations

- **Exclusive Or**
  
<table>
<thead>
<tr>
<th>ALUInstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>val fc = 11</td>
</tr>
<tr>
<td>Result = r \text{ xor } m</td>
</tr>
<tr>
<td>\Delta B = ()</td>
</tr>
</tbody>
</table>

  Returns the exclusive or of the two input words.

- **And**
  
<table>
<thead>
<tr>
<th>ALUInstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>val fc = 9</td>
</tr>
<tr>
<td>Result = r \text{ and } m</td>
</tr>
<tr>
<td>\Delta B = ()</td>
</tr>
</tbody>
</table>

  Returns the logical and of the two inputs.

- **Or**
  
<table>
<thead>
<tr>
<th>ALUInstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>val fc = 10</td>
</tr>
<tr>
<td>Result = r \text{ or } m</td>
</tr>
<tr>
<td>\Delta B = ()</td>
</tr>
</tbody>
</table>

  Returns the logical or of the two inputs.

```
LogicalOp = ( Or \text{ v And v ExclusiveOr } )
```

The three logical operators. There are $15 \times 4 = 3 \times 100$ possible logical operations.
4.9 Load Instruction

```
MonadicInstruction
IndexedAddressing
DB : RegName = Word
val fc = 13
```

The operation is a monadic or load instruction. There is no register select, the only operand comes from the a input to the ALU. The register select field sl is used to determine which operation is performed.

```
LoadRegister
MonadicInstruction
val s1 = 0
Result = Mem(offs trim 20)
DB = ()
io = 0
```

Simply load the register with a value from a memory location.

```
LoadAndNegateRegister
MonadicInstruction
val s1 = 1
Result = zero minus (Mem(offs trim 20))
DB = ()
io = 0
```

Load and find the 2's complement of the value from a memory location. There is no check to see if there has been an overflow as this is done in a later error schema.

```
LoadEffectiveAddress
MonadicInstruction
val s1 = 2
Result = offs
DB = ()
val s2 = 12
io = 0
```

Load the address determined by the addressing mode into the result.
Load in a word from PERipheral space.

`LoadOp = LoadRegister ` LoadAndNegateRegister ` LoadEffectiveAddress ` InputFromPERI`

One of the four load operations. There are $15 \times 1 \times 4 = 60$ possible operations.

`ALU = LogicalOp ` ArithmeticOp ` LoadOp`

An ALU operation. At present there are $300 + 180 + 60 = 540$ operations defined.
Load the result from the ALU into the A register and set the IA flag false to show that there is valid data in the A register. Also set the B and IB flags if they should be set by this operation.

Load the result from the ALU into the X register and set the IX flag false to show that there is valid data in the A register. Also set the B and IB flags if they should be set by this operation.

Load the result from the ALU into the Y register and set the IY flag false to show that there is valid data in the A register. Also set the B and IB flags if they should be set by this operation.

Load the result from the ALU into the Z register and set the IZ flag false to show that there is valid data in the A register. Also set the B and IB flags if they should be set by this operation.

Load one of the four general purpose registers. There are 540 \* 4 = 2160 possible operations. The two other function codes fc = 13, fc = 14 will give another 15 \* 4 \* 2 \* 4 = 480 operations. This means that in total there are 2640 data operations possible.
4.11 Exception Handling for ALU Operations

**SignedAddOverflow**

- **ALUInstruction**
- **ErrorInstruction**

\[
\text{val } fc = 5 \\
\text{(r overflow m) } = 1 \\
\Delta\text{Mem} = \emptyset
\]

An overflow has occurred on a signed add.

**SignedSubtractUnderflow**

- **ALUInstruction**
- **ErrorInstruction**

\[
\text{val } fc = 7 \\
\text{(r underflow m) } = 1 \\
\Delta\text{Mem} = \emptyset
\]

An underflow has occurred on a signed subtract.

**SignedMultiplyOverflow**

- **ALUInstruction**
- **ErrorInstruction**

\[
\text{val } fc = 12 \\
\text{(r overflow m) } = 1 \\
\Delta\text{Mem} = \emptyset
\]

An overflow has occurred on a signed multiply.

**LoadAndNegateRegisterOverflow**

- **MonadicInstruction**
- **ErrorInstruction**

\[
\text{val } s1 = 1 \\
\text{m = Mem(offs trim 20)} \\
\text{(zero underflow m) } = 1 \\
\Delta\text{Mem} = \emptyset
\]

An underflow has occurred when loading and negating a register. This means that the value which was loaded must have been maxneg.
LoadEffectiveAddressError →
  MonadInstruction
  ErrorInstruction

val s1 = 2
val s2 > 12
ΔMem = ()

Illegal operation: if s2 > 12 then it is immediate or register addressing, i.e. there is no 'effective address'.

InputFromPERIError →
  MonadInstruction
  ErrorInstruction

val s1 = 3
val s2 > 3
ΔMem = ()

The operation is an input from PERI, but the addressing mode is not global.

IllegalAddress →
  MonadInstruction
  ErrorInstruction

val s1 ≠ 2
invalid offs = 1
ΔMem = ()

The operation has been defined as a load address but the address is not legal.

MonadError & LoadEffectiveAddressError v
  InputFromPERIError

MonadError & MonadError v
  MonadError & IllegalAddress

MonadError & MonadError v
  MonadError & LoadAndNegateRegisterOverflow

Needed to cope with two errors in the same instruction. A Load Effective Address Error will be noticed before an Illegal Input Address error which will be noticed before a Load and negate register overflow.

ArithError & SignedAddOverflow v
  SignedSubtractUnderflow v
  SignedMultiplyOverflow v
  MonadError
The Errors which can occur during ALU operations.
4.12 Jumps and Calls

The instruction is a control instruction.

The framing schema for a jump or a call. Destination is the location to call or branch to. Note three types of jump: absolute or Program Counter relative forwards or backwards.

Unconditional Jump

Unconditional jump. P is loaded with the value of destination.

Jump if the E (error) flag is set. Set all of the Illegal Register flags to false?
Jump if the B flag is set.

Jump if the B flag is not set.

Jump if the attention input to the ViperZ microprocessor is set.

Jump if the attention input to the ViperZ microprocessor is not set.

FailedJumpCondition

ControlInstruction

EViperZ
If the jump condition is false, then Viper2 state the same (apart from the program counter increment).

DecrementAndJumpOnNotZero

<table>
<thead>
<tr>
<th>DestinationSelect</th>
<th>RegisterSelect</th>
<th>ΔPC</th>
<th>RegName</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ΔPC</td>
<td>RegName</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{val } fc = 6
\]
\[
\text{Result } = r \text{ minus one}
\]
\[
(\text{Result } \not= \text{ zero}) \rightarrow (\text{ΔPC } = (\text{P} + (\text{Destination trim 20})))
\]
\[
(\text{Result } = \text{ zero}) \rightarrow (\text{ΔPC } = (0))
\]
\[
(\text{val } s1 = 0) \rightarrow (\text{ΔReg } = (\text{X-Result.IA-False } \oplus \text{ΔPC})
\]
\[
(\text{val } s1 = 1) \rightarrow (\text{ΔReg } = (\text{X-Result.IX-False } \oplus \text{ΔPC})
\]
\[
(\text{val } s1 = 2) \rightarrow (\text{ΔReg } = (\text{Y-Result.IY-False } \oplus \text{ΔPC})
\]
\[
(\text{val } s1 = 3) \rightarrow (\text{ΔReg } = (\text{ZI-Result.IZ-False } \oplus \text{ΔPC})
\]
\[
\text{ΔMem } = (\omega)
\]

Decrement the selected register, and jump if it is not zero.

CallInstruction

<table>
<thead>
<tr>
<th>DestinationSelect</th>
<th>ΔFlags</th>
<th>RegName</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ΔFlags</td>
<td>RegName</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{TopOfCallFrame.BottomOfNewWorkspace } : \text{ Data}
\]
\[
\text{BottomOfCallFrame.ProgramStatusWord } : \text{ Data}
\]

\[
\text{BottomOfCallFrame } = ((\text{Reg } F) \text{ pad 32 }) \text{ plus } ((\text{Reg } S) \text{ pad 32 })
\]
\[
\text{TopOfCallFrame } = \text{ BottomOfCallFrame } \text{ plus one}
\]
\[
\text{ProgramStatusWord } = ((\text{Reg } P) \text{ pad 32 } \oplus (20-\text{val } (\text{Reg Trust})))
\]
\[
\text{BottomOfNewWorkspace } = \text{ TopOfCallFrame } \text{ plus one}
\]
\[
\text{ΔMem } = ((\text{BottomOfCallFrame trim 20 }, \text{Reg } F),
\]
\[
\text{TopOfCallFrame trim 20 }, \text{ProgramStatusWord})
\]
\[
\text{ΔReg } = (\text{P} + (\text{BottomOfNewWorkspace})
\]
\[
\text{P} + (\text{Destination trim 20}), \text{Postcall } = \text{True}\)
\]
\[
\text{ΔFlags}
\]

The Call instruction. Set up the link frame on the stack, set the frame pointer to point to the bottom of the new workspace, set the postcall register to True to ensure that the next instruction is an Enter and load in the new value for the program counter. The value in the error flags may also alter if there is a call on Error instruction. The link frame consists of two data words.

The first word is placed in the location above the top of the previous stack frame and is loaded with the old frame pointer. The second word is placed in the location above the first word. This holds the return program counter as well as the old value of the trust bit.

UnconditionalCall

<table>
<thead>
<tr>
<th>CallInstruction</th>
<th>ΔFlags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ΔFlags</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{val } fc = 8
\]
\[
\text{ΔFlags } = (\omega)
\]

45
Unconditional jump. P is loaded with the value of destination.

```
val fc = 9
val (Reg E) = 1
bFlags = [(A=False), (X=False), (Y=False),
         (IA=False), (IB=False), (E=False)]
```

Call if the E (error) flag is set. Set all of the illegal register flags to false.

```
val fc = 10
val (Reg B) = 1
bFlags = ()
```

Call if the B flag is set.

```
val fc = 11
val (Reg B) = 0
bFlags = ()
```

Call if the B flag is not set.

```
val fc = 12
attention = 1
bFlags = ()
```

Call if the attention input to the ViperZ microprocessor is set.

```
val fc = 13
attention = 0
bFlags = ()
```

Call if the attention input to the ViperZ microprocessor is not set.
If the call condition is false, then ViperZ state the same (apart from the Program counter increment).
4.13 Copy Instruction

CopyFromRegisterToGeneralPurposeRegister

ControlInstruction
ad : N

val fc = 7
val fa = 0
ad = (val addr) mod 16
(ad = 0) → (Result = Reg A)
(ad = 1) → (Result = Reg X)
(ad = 2) → (Result = Reg Y)
(ad = 3) → (Result = Reg Z1)
(ad = 4) → (Result = (Reg P) pad 32)
(ad = 5) → (Result = (Reg F) pad 32)
(ad = 6) → (Result = (Reg U) pad 32)
(ad = 7) → (Result = (Reg Watchdog) pad 32)
(ad = 8) → (Result = Reg D)

Copy from a register to a general purpose register.

CopyToGeneralPurposeRegister

CopyFromRegisterToGeneralPurposeRegister

(val s1 = 0) → (bReg = (A=Result,IA=False))
(val s1 = 1) → (bReg = (X=Result,IX=False))
(val s1 = 2) → (bReg = (Y=Result,IY=False))
(val s1 = 3) → (bReg = (Z1=Result,IZ=False))

Place value in general purpose register.

CopyFromGeneralPurposeRegisterToRegister

ControlInstruction
RegisterSelect
ad : N

val fc = 7
val fa = 1
Reg Trust = True
ad = (val addr) mod 16
(ad = 0) → (bReg = (Awr,IA=False))
(ad = 1) → (bReg = (Xwr,IX=False))
(ad = 2) → (bReg = (Ywr,IY=False))
(ad = 3) → (bReg = (Z1wr,IZ=False))
(ad = 4) → (bReg = (Pwr trim 20))
(ad = 5) → (bReg = (Wwr trim 20,NOlimt-True,
NoSize-True,NoStack=False))
(ad = 6) → (bReg = (Swr trim 20,NOSize=False))
(ad = 7) → (bReg = (Uwr trim 20,NOLimit=False))
(ad = 8) → (bReg = (Watchdog=(r trim 16),WE=False))
(ad = 9) → (bReg = (Dwr))
Copy a value from a general purpose register to a special register.
4.14 Enter and Return

Enter

\[
\text{ControlInstruction}
\]
\[
\begin{align*}
\text{val fc} & = 14 \\
\text{(val (Reg F)) + (val addr) + 2} & \leq \text{(val (Reg U))} \\
\text{Reg Postcall} & = \text{True} \\
\text{(val fo = 0)} & \Rightarrow (\Delta \text{Reg} = (S-addr, \text{Postcall} = \text{False})) \\
\text{(val fo = 1)} & \Rightarrow (\Delta \text{Reg} = (S-addr, \text{Trust} = \text{False}, \text{Postcall} = \text{False})) \\
\text{(val fo = 2)} & \Rightarrow (\Delta \text{Reg} = (S-addr, \text{Trust} = \text{True}, \text{Postcall} = \text{False}))
\end{align*}
\]

The Enter Instruction. This must be executed immediately after a call instruction. If it is called at any other time it will generate an error. The enter instruction sets up the frame size required by the routine, after checking that at least 2 words of memory are free at the top of the new frame to accommodate a call instruction in the new routine. It also sets up the trusted ness of the routine. Finally the postcall bit is reset.

Return

\[
\text{CallInstruction}
\]
\[
\begin{align*}
\text{TopOfCallFrame} & = (\text{Reg F}) \text{ minus one} \\
\text{ProgramStatusWord} & = \text{Mem(TopOfCallFrame)} \\
\text{BottomOfCallFrame} & = \text{TopOfCallFrame minus one} \\
\text{BottomOfNewWorkspace} & = \text{Mem(BottomOfCallFrame)} \\
\Delta \text{Reg} & = (F-\text{(BottomOfNewWorkspace)}, \\
\text{P} & = (\text{ProgramStatusWord trim 20}), \\
\text{Trust} & = (\text{ProgramStatusWord 20}) \\
\text{S} & = (\text{BottomOfCallFrame minus} \\
\text{(BottomOfNewWorkspace trim 20)})
\end{align*}
\]

The Return from subroutine command. This basically undoes the call command. The frame pointer (F), program counter and trust bit are reloaded from the link frame. The value in the frame size register is calculated and loaded back in.

Copies

\[
\begin{align*}
\text{CopyFromGeneralPurposeRegisterToRegister} & \quad \text{v} \\
\text{CopyToGeneralPurposeRegister} & \quad \text{v}
\end{align*}
\]

The two copy commands. This covers \(1 \times 4 \times 1 \times 2 = 8\) operations.

Jumps

\[
\begin{align*}
\text{UnconditionalJump} & \quad \text{v} \\
\text{JumpIfError} & \quad \text{v} \\
\text{JumpIfAttentionSet} & \quad \text{v} \\
\text{JumpIfBNotSet} & \quad \text{v} \\
\text{DecrementAndJumpOnNotZero} & \quad \text{v} \\
\text{JumpIfAttentionNotSet} & \quad \text{v} \\
\text{JumpIfBSet} & \quad \text{v}
\end{align*}
\]

The seven jump commands. This covers \(1 \times 4 \times 7 = 28\) operations.
Call & UnconditionalCall & CallIfError & CallIfBSet & CallIfBNotSet & CallIfAttentionSet & CallIfAttentionNotSet

Calls & Call & FailedCallCondition

The six call commands. This covers $1 \times 4 \times 6 \times 3 = 72$ operations.

Control & Calls & Jumps & Copies & Enter & Return

The control operations. There are $8 + 84 + 72 + 1 = 4 \times 2 = 3 = 188$ operations.
4.14 Illegal Calls and Jumps

**IllegalJump**

- **Jump**
- **ErrorInstruction**

*invalid Destination = 1*

The operation is a jump but the destination is not in memory space.

**IllegalJumpCondition**

- **Jumps**
- **ErrorInstruction**

*Reg IB = True*

*val fc = 2 v (val fc = 3)*

The jump is dependant on B, but B has not been set.

**IllegalJumps**

- **a IllegalJump**
- **v IllegalJumpCondition**

**IllegalCallError**

- **ControlInstruction**
- **ErrorInstruction**
- **DestinationSelect**

TopOfCallFrame, BottomOfNewWorkspace : Data

BottomOfCallFrame, ProgramStatusWord : Data

*val fc ≥ 8 A (val fc ≤ 13)*

BottomOfCallFrame = ((Reg F) pad 32) plus

((Reg S) pad 32)

TopOfCallFrame = BottomOfCallFrame plus one

ProgramStatusWord = (Reg P) pad 32 + (20-\text{val(Reg Trust)})

BottomOfNewWorkspace = TopOfCallFrame plus one

Framing schema for Call errors.

**IllegalDestination**

- **IllegalCallError**

*invalid Destination = 1*

*\delta Mem = {}*

The operation is a call but the destination is not in memory space.
IllegalBottomOfCallFrame
IllegalCallError

invalid BottomOfCallFrame = 1
\Delta Mem = ()

The bottom of the call space is not in memory.

IllegalTopOfCallFrame
IllegalCallError

invalid TopOfCallFrame = 1
\Delta Mem = ((BottomOfCallFrame trim 20) - (Reg F))

The top of the call space is not in memory. This is only noticed after the first write to memory has been made.

IllegalBottomOfNewWorkspace
IllegalCallError

invalid BottomOfNewWorkspace = 1
\Delta Mem = ((BottomOfCallFrame trim 20) - (Reg F), (TopOfCallFrame trim 20 ln(ProgramStatusWord))

The bottom of the new workspace is not in memory. This is only noticed after the first two writes to memory have been made.

StackNotSet
IllegalCallError

(Reg NoStack = True) \lor (Reg NoSize = True)
\Delta Mem = ()

A call has been made with the stack not set.

IllegalCalls * IllegalDestination ` v
IllegalBottomOfCallFrame ` v
IllegalTopOfCallFrame ` v
IllegalBottomOfNewWorkspace ` v
StackNotSet

All of the Illegal Call schemes.
4.15 Illegal Copy

```
4.15.1 Illegal Copy

CopyError1

Control Instruction
Error Instruction

val fc = 7
val fa = 0
\Delta Mem = \{\}
 Officials = \{\}
\begin{align*}
  (\text{val addr} = 0) & \wedge (\text{Reg IA} = \text{True}) \\
  (\text{val addr} = 1) & \wedge (\text{Reg IX} = \text{True}) \\
  (\text{val addr} = 2) & \wedge (\text{Reg IY} = \text{True}) \\
  (\text{val addr} = 3) & \wedge (\text{Reg NoStack} = \text{True}) \\
  (\text{val addr} = 4) & \wedge (\text{Reg NoSize} = \text{True}) \\
  (\text{val addr} = 5) & \wedge (\text{Reg NoLimit} = \text{True}) \\
\end{align*}

Attempt to copy invalid register.

CopyError2

Control Instruction
Error Instruction

val fc = 7
val fa = 1
\Delta Mem = \{\}
Officials = \{\}
\begin{align*}
  (\text{val sl} = 0) & \wedge (\text{Reg IA} = \text{True}) \\
  (\text{val sl} = 1) & \wedge (\text{Reg IX} = \text{True}) \\
  (\text{val sl} = 2) & \wedge (\text{Reg IY} = \text{True}) \\
\end{align*}

Attempt to copy invalid register.

IllegalCopy

Control Instruction
Error Instruction

val fc = 7
val fa = 1
\Delta Mem = \{\}
Officials = \{\}
\begin{align*}
  (\text{val sl} = 0) & \wedge (\text{Reg IA} = \text{True}) \\
  (\text{val sl} = 1) & \wedge (\text{Reg IX} = \text{True}) \\
\end{align*}

Attempt to copy to protected register, in untrusted mode.

IllegalCopies & CopyError1 v CopyError2 v IllegalCopy

Error in copying from register to register.
```
<table>
<thead>
<tr>
<th>LimitNotSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlInstruction</td>
</tr>
<tr>
<td>ErrorInstruction</td>
</tr>
</tbody>
</table>

```plaintext
val fc = 14
ΔMem = Ø
```

Limit is not set in enter instruction.
4.16 Postcall and Enter Errors

**PostcallNotSet**
- ControlInstruction
- ErrorInstruction
- val fc = 14
- Reg Postcall = False
- ΔMem = {}  

Postcall is not set and Enter has been found, i.e., Enter has occurred somewhere other than at the start of a subroutine.

**EnterNotFound**
- ControlInstruction
- ErrorInstruction
- val fc = 14
- Reg Postcall = True
- ΔMem = {}  

Postcall is set and Enter has not been found, i.e., Enter has not occurred at the start of a subroutine.

**StackOverflow**
- ControlInstruction
- ErrorInstruction
- val fc = 14
- (val (Reg F)) + (val addr) + 2 > (val (Reg U))
- Reg Postcall = True
- ΔMem = {}  

The stack cannot accommodate the present frame.
4.17 Write Operations

- GlobalWrite

\[ \text{Write Instruction} \]
\[ \text{RegisterSelect} \]
\[ \text{val sz} = 15 \]
\[ \text{val fq} = 3 \]
\[ \text{val fc} \leq 11 \]
\[ \text{Reg} = () \]

Write instruction. Note fc > 11 is an illegal op code.

- LocalStackFrameWrite

\[ \text{Write Instruction} \]
\[ \text{val fch} = 0 \]
\[ \text{base} = \text{addr pad 32} \]
\[ \text{io} = 0 \]
\[ \text{Reg Trust} = \text{True} \]

Write to local stack frame.

- OutputToPERI

\[ \text{Write Instruction} \]
\[ \text{val fch} = 2 \]
\[ \text{base} = \text{addr pad 32} \]
\[ \text{io} = 1 \]

Output to PERipheral.

The three addressing modes.

- Write

\[ \text{Write Base} \]
\[ \text{GlobalWrite} \]
\[ \text{LocalStackFrameWrite} \]
\[ \text{OutputToPERI} \]

\[ \text{WriteBase} \]
\[ \text{GlobalWrite} \]
\[ \text{LocalStackFrameWrite} \]
\[ \text{OutputToPERI} \]

The three addressing modes.
Write to the location specified. Either absolute addressing or indexed addressing. Write has \( 1 \times 4 \times 1 \times 12 = 48 \) operations.
4.18 Write Errors

WriteError
WriteInstruction
ErrorInstruction

(val fcl = 0) A (offs = base) v
(val fcl = 1) A (offs = base plus (Reg X)) v
(val fcl = 2) A (offs = base plus (Reg Y)) v
(val fcl = 3) A (offs = base plus (Reg Z))

Write error framing schema.

IllegalIndex
WriteInstruction
ErrorInstruction

(val fcl = 1) A (Reg IX = True) v
(val fcl = 2) A (Reg IY = True) v
(val fcl = 3) A (Reg IZ = True)

The index register specified is illegal.

GlobalWriteError
WriteError
val fch = 0
Mem = {}
invalid offs = 1

The write location is not in the memory space.

StackFrameWriteError
WriteError
val fch = 1
Mem = {}
invalid offs = 1
(val offs < (val (Reg F)) v
(val offs > (val (Reg F)) + (val (Reg 5)))

The write location is not in the stack frame.

GlobalOutputError
WriteError
val fch = 2
Mem = {}
invalid offs = 1

The output location is not in the memory space.
IllegalWriteAddress & GlobalWriteError
StackFrameWriteError
GlobalOutputError
IllegalIndex

The Write Errors.
4.19 Other ViperZ Errors

- **WatchdogTimeout**
  - **Reg WE** = True
  - **Reg Trust** = False

The watchdog timer has timed out, and ViperZ is in untrusted mode.

- **IllegalOpCode**
  - **val s2 = 15**
  - **((val fc = 6) \& (val fq = 2)) \& ((val fq = 3) \& (val fc z12))**

An illegal Op code. There are $1 \times 4 \times 1 \times 1 + 1 \times 4 \times 1 \times 4 = 20$ possibilities.

- **Viper2_Error** a IllegalP
  - UnsetAddressingRegister
  - RegisterSelectInvalid
  - IllegalReadAddress
  - ArithError
  - IllegalJumps
  - IllegalCopies
  - LimitNotSet
  - EnterNotFound
  - StackOverflow
  - PostcallNotSet
  - IllegalCalls
  - IllegalWriteAddress
  - WatchdogTimeout
  - IllegalOpCode

The Viper2 Error conditions.

- **arb : Word \rightarrow Word**

  (w1, w2) \rightarrow (w1 = w2 \& w1 = arb w2)

The arb function, has no relationship between input and output words.

- **TrustedError**
  - **Viper2_Error**
  - **Reg Trust** = True
  - **stop' = 1**

Error in trusted state, machine stops.
UntrustedError
ViperZ_Error
CallInstruction

Reg Trust = False
TopOfCallFrame = (Reg F) minus one
ProgramStatusWord = Mem(TopOfCallFrame)
BottomOfCallFrame = TopOfCallFrame minus one
BottomOfNewWorkspace = Mem(BottomOfCallFrame)
6Reg = {A=arb (Reg A)),IA=True,
X=arb (Reg X)),IX=True,
Y=arb (Reg Y)),IY=True,
Z1=arb (Reg Z1)),IZ=True,
B=arb (Reg B)),IB=True,
E=True,
F=Mem(BottomOfNewWorkspace),
Pu=ProgramStatusWord trim 20),
Trust=(ProgramStatusWord 20),
Sw=BottomOfCallFrame minus
(BottomOfNewWorkspace trim 20))

stop' = 0

Error in untrusted state. Set all Error flags true and return from subroutine.

ViperZErrore UntrustedError y TrustedError
4.20 The Viper Top Level Specification

\[
\begin{align*}
\text{NotStopped} & \quad \Delta \text{Viper2} \\
\text{stop'} & = 0 \\
\end{align*}
\]

\text{ViperOK} \land \text{Compare} \lor \text{ALUOp} \lor \text{Control} \lor \text{Write}

Viper2 has successfully completed an operation. There are 1200 + 2040 + 180 + 40 + 20 = 4096 possible operations, i.e., all Op codes accounted for.

\text{OKState} \land (\neg \text{Viper2_Errors}) \land \text{ViperOK} \land \text{NotStopped}

\text{NextState} \land (\neg \text{Viper2_Errors}) \land \text{OKState} \lor \text{Stopped} \lor \text{Reset}

The next state of the Viper2 machine.

5 Conclusions

This document gives an initial specification of the Viper2 in Z. It has been shown that Z provides a higher level of specification than that written in HOL. It has also demonstrated that it is a useful language to produce a high level specification of a microprocessor.

This specification was completed before the HOL specification was complete and so no attempt was made to ensure conformity between the two.
6 Acknowledgements

W.J. Cullyer who produced the HDL specification.
C. Pygott and J. Kershaw for the design of the ViperZ.
C. O'Halloran for his help with the Z editor and type checker.
J. Wiseman for suggesting modifications.
A. Passa for turning the Z document into a Memorandum.

7 References

1. Kemp D.H. Specification of Viper1 in Z
2. Cullyer W.J. Viper2 Microprocessor: Formal Specification
   To be published.
   Instruction Set.
   Prentice-Hall International series in
   computer science, 1987.
As a continuation of the use of the specification language Z which was used to specify the Viper 1 microprocessor this paper covers the specification of the Viper 2. This was completed before the definitive HOL specification was complete, therefore there is no proof of correspondence between the two. Using Z did highlight inconsistencies in the HOL specification that may not have appeared until later in the specification.