RSRE
MEMORANDUM No. 4217

ROYAL SIGNALS & RADAR ESTABLISHMENT

SPECIFICATION OF VIPER2 IN Z

Author: D H Kemp

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

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.

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 [1]. 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 [2]. 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. Bowan's **Z** specification of the M6800 microprocessor[3] 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 [4] 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.

†*Higher Order Logic (HOL) is a design tool developed at the Cambridge Computing Laboratory.*

2 Basic Functions

2.1 Bits and Words

```
Bit ≙ {0,1}

Word ≙ { w:ℕ⇸Bit | #w>0 ∧ 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, ie the result of w(n) (the word w acting on the value n) gives the n+1th Bit of the word w.

```
| LSB,MSB : Word → Bit
|----------------------
| ∀ w : Word •
| LSB w = w 0
| MSB w = w #w-1
```

Find the most and least significant bits of the word.

```
| val : Word → ℕ
|------------------------------------------
| ∀ w : Word •
|   (#w=1) ⇒ ( val w = LSB w )
|   (#w>1) ⇒ ( val w = LSB w + 2 * val(succ;w))
```

val returns the natural number represented by the word. Note succ;w gives the effect of a Right shift, ie divide by two, on the word. ie if succ;w is applied to n then first succ n is calculated, and then w of n+1 is calculated ie the n+2th Bit is returned rather than the n+1th one.

```
| pred : ℕ₁ → ℕ
|------------------------
| ∀ n : ℕ • pred n = n - 1
```

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

```
| (_set_) : (Word×Bit) → Word
|------------------------------
| ∀ w : Word; b : Bit •
|   w set b = w;{(0↦b),(1↦b)}
```

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

```
| maxval : Word → ℕ
|------------------------
| ∀ w : Word •
|   maxval w = val(w set 1)
```

$$\vdash \neg\ (\ \exists\ w : Word \bullet ((\ val\ w) > maxval\ w)\ )$$

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

$$\begin{array}{|l}
wrd : \mathbb{N}_1 \rightarrow (\ \mathbb{N} \rightarrow Word\ ) \\
\hline
\forall\ size : \mathbb{N}_1;\ valu : \mathbb{N};\ w : Word \bullet \\
\quad (wrd\ size\ valu = w\ ) \Leftrightarrow \\
\quad\quad ((\#w = size) \wedge \\
\quad\quad\quad (val\ w = valu\ mod\ succ(maxval\ w)))
\end{array}$$

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

$$\begin{array}{|l}
(\_\hat{}\_) : (\ Word \times Word) \rightarrow Word \\
\hline
\forall\ w1,w2 : Word \bullet \\
\quad w1\hat{}w2 = w1 \quad \cup\ (\ pred^{\#w1}\ ;\ w2\ )
\end{array}$$

$$\vdash \forall\ w1,w2 : Word \bullet \#(w1\hat{}w2) = \#w1 + \#w2$$

Concatinate two words together.

2.2 Bitwise Functions

```
| not : Bit ⤖ Bit
|──────────────────
| not = {0↦1,1↦0}
```

Generate the logical inverse of the input bit.

```
| (_•_),(_+_),(_⊕_) : (Bit×Bit)→Bit
|──────────────────────────────────────────
| (_+_) = {(0,0)↦0,(0,1)↦1,(1,0)↦1,(1,1)↦1}
| (_⊕_) = {(0,0)↦0,(0,1)↦1,(1,0)↦1,(1,1)↦0}
| (_•_) = {(0,0)↦0,(0,1)↦0,(1,0)↦0,(1,1)↦1}
```

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

Viper_aux_1 keeps •,+,⊕,not,^,wrd,maxval,set,pred,val,LSB,MSB,
Word,Bit

2.3 Logical fuctions on words

```
| wnot : Word ⤖ Word
|---------------------------
| ∀ w : Word •
|   wnot w = w ⨾ not
```

Generate the inverse of the input word.

```
WordPair ≙
    { w : ℕ ⇸ (Bit×Bit) | #w>0 ∧ dom w = 0 .. ((#w)-1) }
```

```
| (_pair_) : (Word×Word)→WordPair
|-------------------------------------------------------
| ∀ w1,w2 : Word •
|   w1 pair w2 =
|     { i : ℕ | i ∈ dom w1 ∩ dom w2 • i ↦ ( w1 i,w2 i) }
```

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

```
| (_and_),(_or_),(_exor_) : (Word×Word)→Word
|---------------------------------------------
| ∀ w1,w2 : Word •
| w1 and  w2 = ((w1 pair w2 ) ⨾ (_•_))
| w1 or   w2 = ((w1 pair w2 ) ⨾ (_+_))
| w1 exor w2 = ((w1 pair w2 ) ⨾ (_⊕_))
```

Standard wordwise logical functions.

```
| (_<<_) : (Word×Bit) → Word
|---------------------------------------------
| ∀ w : Word; b : Bit •
|   w << b = ( {#w} ⩤ (pred ⨾ w) ) ∪ {0↦b}
```

```
| (_>>_) : (Bit×Word) → Word
|---------------------------------------------
| ∀ w : Word; b : Bit •
|   b >> w = {((#w)-1) ↦ b} ∪ (succ ⨾ w )
```

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

2.4 Arithmetic Functions

$$
\begin{array}{|l}
value : Word \rightarrow \mathbb{Z} \\
\hline
\forall w : Word \bullet \\
\quad ((MSB\ w = 1) \wedge value\ w = val\ w - succ\ (maxval\ w\ )) \vee \\
\quad ((MSB\ w = 0) \wedge value\ w = val\ w\ )
\end{array}
$$

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.

$$
\begin{array}{|l}
maxpos, maxneg : Word \rightarrow \mathbb{Z} \\
\hline
\forall w1, w2 : Word \mid \#w1 = ((\#w2)+1)\ \bullet \\
\qquad maxpos\ w1 = maxval\ w2 \\
\qquad maxneg\ w1 = (maxval\ w2)\ - (maxval\ w1)
\end{array}
$$

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

$$
\begin{array}{|l}
(\_signextend\_) : (Word \times \mathbb{N}_1) \rightarrow Word \\
\hline
\forall w1, w2 : Word;\ length : \mathbb{N}_1 \mid \\
\qquad\qquad\qquad (length \geq \#w1) \wedge (\#w2 = length)\ \bullet \\
\quad (w1\ signextend\ length) = (w2\ set\ (MSB\ w1)) \bullet w1
\end{array}
$$

Sign extends the word to the length specified.

$$
\begin{array}{|l}
(\_pad\_) : (Word \times \mathbb{N}_1) \rightarrow Word \\
\hline
\forall w1, w2 : Word;\ length : \mathbb{N}_1 \mid \\
\qquad\qquad\qquad (length \geq \#w1) \wedge (\#w2 = length)\ \bullet \\
\quad (w1\ pad\ length) = (w2\ set\ 0) \bullet w1
\end{array}
$$

Pad out a word to the new word length with zeros

$$
\begin{array}{|l}
(\_trim\_) : (Word \times \mathbb{N}_1) \rightarrow Word \\
\hline
\forall w : Word;\ length : \mathbb{N}_1 \mid length \leq \#w\ \bullet \\
\quad w\ trim\ length = (0 \mathinner{..} length) \lhd w
\end{array}
$$

Trim a word down to the new word length. Note, use the above with caution, as it simply returs a word with the top bits 'trimmed' off. No check is made to ensure that the value of the word has not changed.

```
| (_plus_) : (Word×Word) → Word
|--------------------------------------------------------------
| ∀ w1,w2,w3 : Word | (#w1) = ((#w2)+1) ∧ (#w2) =(#w3)  •
|   (w2 plus w3 = (w1 trim #w2))
|                   ⇔ (value w1)=(value w2)+(value w3)
```

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.

```
| (_times_) : (Word×Word) → Word
|--------------------------------------------------------------
| ∀ w1,w2,w3 : Word | (#w1) = ((#w2)*2) ∧ (#w2) =(#w3)  •
|   (w2 times w3 = (w1 trim #w2))
|                   ⇔ (value w1)=(value w2)*(value w3)
```

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.

```
| (_minus_) : (Word×Word) → Word
|--------------------------------------------------------------
| ∀ w1,w2,w3 : Word | (#w1) = ((#w2)+1) ∧ (#w2) =(#w3)  •
|   (w2 minus w3 = (w1 trim #w2))
|                   ⇔ (value w1)=(value w2)-(value w3)
```

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.

```
| (_carry_) : (Word×Word) → Bit
|--------------------------------------------------------------
| ∀ w1,w2 : Word •
|   (w1 carry w2 = 1) ⇔ ((val w1) + (val w2) > maxval w1)
```

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

```
| (_mcarry_) : (Word×Word) → Bit
|--------------------------------------------------------------
| ∀ w1,w2 : Word •
|   (w1 mcarry w2 = 1) ⇔ ((val w1) + (val w2) > maxval w1)
```

Top level specification of carry for multiplication.

```
| (_borrow_) : (Word×Word) → Bit
|--------------------------------------------------------------
| ∀ w1,w2 : Word •
|   (w1 borrow w2 = 1) ⇔ ( (val w1) < (val  w2) )
```

Top level specification of Borrow.

```
| (_overflow_) : (Word×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, ie overflow when the sum is greater than the largest positive value which can be held, or less than the largest negative number.

```
| (_moverflow_) : (Word×Word) → Bit
|------------------------------------------------
| ∀ w1,w2 : Word | #w1 = #w2 •
|   (w1 moverflow w2 = 1) ⇔
|          ( ( (value w1) + (value w2) > maxpos w1  ) v
|            ( (value w1) + (value w2) < maxneg w2  ) )
```

Top level specification of overflow for multiplication.

```
| (_underflow_) : (Word×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×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×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

$Word_{64} \triangleq \{ w : Word \mid \#w = 64 \}$
-- For Double length integers

$Word_{32} \triangleq \{ w : Word \mid \#w = 32 \}$
-- For Data words

$Word_{20} \triangleq \{ w : Word \mid \#w = 20 \}$
-- For Address words

$Word_{4} \triangleq \{ w : Word \mid \#w = 4 \}$
-- For the function select

$Word_{3} \triangleq \{ w : Word \mid \#w = 3 \}$
-- For the destination select

$Word_{2} \triangleq \{ w : Word \mid \#w = 2 \}$
-- For the register and memory select

$Word_{1} \triangleq \{ w : Word \mid \#w = 1 \}$
-- For the comparison select and flags

$Address \triangleq Word_{20}$

$Data \triangleq Word_{32}$

$Flag \triangleq Word_{1}$

```
┌ Values ──────────────
│ one,zero : Word
│ True,False : Flag
├──────────────────────
│ 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.

```
┌─Memory──────────────────────────
│ Mem       : Address → Data
│ PERIspace : Address → Data
│ RAMspace  : Address → Data
│ io        : Bit
├─────────────────────────────────
│ (io = 0) ⇒ (Mem = RAMspace)
│ (io = 1) ⇒ (Mem = PERIspace)
└─────────────────────────────────
```

If io is zero then all memory reads are from the RAM space. If io is one then all of the reads are from the PERIspace.

```
┌─ΔMemory─────────────────────────────────
│ Memory
│ Memory'
│ δMem : Address ⇸ Data
├─────────────────────────────────────────
│ (io = 0) ⇒ (RAMspace' = RAMspace ⊕ δMem)
│ (io = 1) ⇒ (RAMspace' = RAMspace)
└─────────────────────────────────────────
```

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

```
┌─ΞMemory──
│ ΔMemory
├──────────
│ δMem = ∅
└──────────
```

No change in memory.

3.3 Registers

The specification of the Viper2 registers.

[RegName]

Register ≙ { r: RegName ⇸ Word | #r>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 ∈ $Word_{32}$

Reg X ∈ $Word_{32}$

Reg Y ∈ $Word_{32}$

Reg Z1 ∈ $Word_{32}$

Reg Double ∈ $Word_{64}$

Reg Double = (Reg A) ^ (Reg Z1)

The four general purpose read write registers (note X,Y,Z1 are index registers. The register double is the concatination of the A and Z registers.

AddressRegisters

F,S,U,P : RegName

Reg F ∈ $Word_{20}$

Reg S ∈ $Word_{20}$

Reg U ∈ $Word_{20}$

Reg P ∈ $Word_{20}$

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 (ie 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 Porgram Counter P is the position in memory where the current instruction was read from.

OtherRegisters

D,Watchdog,Temp : RegName

Reg D $\in$ $Word_{32}$
Reg Watchdog $\in$ $Word_{32}$
Reg Temp $\in$ $Word_{32}$

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 $\in$ $Word_1$
Reg Postcall $\in$ $Word_1$
Reg Trust $\in$ $Word_1$

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 $\in$ $Word_1$
Reg IA $\in$ $Word_1$
Reg IX $\in$ $Word_1$
Reg IY $\in$ $Word_1$
Reg IZ $\in$ $Word_1$
Reg IB $\in$ $Word_1$
Reg WE $\in$ $Word_1$
Reg NoStack $\in$ $Word_1$
Reg NoSize $\in$ $Word_1$
Reg NoLimit $\in$ $Word_1$

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, ie IA is true if A has not been loaded since the machine started, or since an error occured. 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 ∧
       ProcessFlags ∧
       OtherRegisters ∧
       AddressRegisters ∧
       GeneralPurposeRegisters
```

The Viper2 register types

```
┌─registers ──────────────
│ Regs
│ Registers : Register
└─────────────────────────
```

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

```
┌─ΔRegisters ─────────────────────────────
│ registers
│ registers'
│ newp         : Address
│ NewWatchdog  : Data
│ NewWE        : Flag
│ δReg         : RegName ⇸ Word
├─────────────────────────────────────────
│ Registers' = Registers ⊕ {P↦newp,WE↦NewWE}
│              ⊕ {Watchdog↦NewWatchdog} ⊕ δReg
└─────────────────────────────────────────
```

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.

```
┌─ΞRegisters ─
│ ΔRegisters
├─────────────
│ δReg = {}
└─────────────
```

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

3.4 Clock

The Viper2 clock is not represented in the HOL specification. A definition is included here for completeness.

```
┌─ Clock ──
│ Clk : ℕ
└────────
```

Clock simply counts up from 0.

```
┌─ ΔClock ──────────
│ Clock
│ Clock'
│ Cycles : ℕ
├─────────────────
│ Clk' = Clk + Cycles
└─────────────────
```

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.

```
┌─ Stop ─────────
│ stop : Bit
└────────────────
```

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

```
┌─ ΔStop ─────────────────
│ ΔRegisters
│ Stop
│ Stop'
│ Values
│ sval : Bit
├─────────────────────────
│ stop  = 0
│ newp  = 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

```
ΔState ≙ ΔMemory ∧
         ΔRegisters ∧
         ΔClock ∧
         ΔStop
```

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 Viper2 Operation Codes

Viper2OpCode

op : $Word_{32}$
s1 : $Word_{2}$
s2 : $Word_{4}$
s2l : $Word_{2}$
s2u : $Word_{2}$
fq : $Word_{2}$
fc : $Word_{4}$
fcl : $Word_{2}$
fch : $Word_{2}$
addr : $Word_{20}$

op = s2 ^ s1 ^ fq ^ fc ^ addr
fc = fch ^ fcl
s2 = s2u ^ s2l

The Viper2 Op code. Op is the op code and is loaded from the address pointed to by the Program Counter. The op code is the concatination 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 Viper2 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.

```
┌─ΔViper2 ──────────────────────────────────────────────
│ ΔState
│ 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 ─────────
│ ΞMemory
│ ΞRegisters
│ Stop
│ Stop'
│ ΔClock
├──────────────────
│ stop  = 1
│ stop' = 1
│ newp  = Reg (P)
└──────────────────
```

The machine has stopped, and cannot restart until there is a Reset.

```
┌─ΞViper2 ─────
│ ΔViper2
│ ΞMemory
│ ΞRegisters
│ ΔStop
└──────────────
```

Viper state unchanged (except P, Watchdog and WE updated)

```
┌─ Reset ──────────────────────────────────────────────
│ ΞMemory
│ ΔRegisters
│ ΔClock
│ ΔStop
│ Viper2Inputs
│ Values
├──────────────────────────────────────────────────────
│ stop' = 0
│ reset = 1
│ val newp = 0
│ δReg = {E↦False,IA↦True,IX↦True,NoStack↦True,NoSize↦True,
│         IY↦True,IZ↦True,IB↦True,Trust↦True,NoLimit↦True,
│         WE↦False,Watchdog↦((Reg Watchdog) set 1)}
└──────────────────────────────────────────────────────
```

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 ─
│ Reset
├─────────────
│ Clk' = 0
└─────────────
```

Machine on start up.

Viper2_machine_state keeps

ΔClock,Stop,ΔStop,ΔViper2,ΞViper2,Viper2INIT,
ΔMemory,ΞMemory,registers,ΔRegisters,Clock,
Address,Data,Flag,Memory,Reset,Reg,$Word_{64}$,
$Word_{32}$,$Word_{20}$,$Word_4$,$Word_3$,$Word_2$,$Word_1$,Stopped,
RegName

This section specifies the inputs to the Viper2 ALU.

```
│ invalid : Word → Bit
├──────────────────────────────────────
│ ∀ w : Word •
│   (invalid w = 1) ⇔ (val w > maxval (wrd 20 0))
```

Returns True if the value is greater than can be held in a 20 bit word.

```
┌─ ErrorInstruction ──
│ ΔViper2
│ ErrorValue : Data
└──────────────────
```

The instruction being executed is illegal. The error code of the particular error is returned in ErrorValue.

```
┌─ IllegalP ─────────────────────────
│ ErrorInstruction
├────────────────────────────────────
│ val (Reg P) = maxval (Reg P)
│ (val s2 = 15) ∨ (val fq = 3) ∨ (val fc ≥ 7)
└────────────────────────────────────
```

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

4 Viper Operations

4.1 Viper2 ALU

This section specifies the vi input to the Viper2 ALU

```
┌─RegisterSelect ──────────────
│ ΔViper2
├──────────────────────────────
│ (val s1 = 0) ⇒ (r = Reg A)
│ (val s1 = 1) ⇒ (r = Reg X)
│ (val s1 = 2) ⇒ (r = Reg Y)
│ (val s1 = 3) ⇒ (r = Reg Z1)
└──────────────────────────────
```

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

```
┌─DataInstruction ─
│ ΔViper2
├──────────────────
│ val s2 ≠ 15
└──────────────────
```

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

4.2 Addressing Modes

```
┌─ GlobalAddressing ─────────
│ DataInstruction
├────────────────────────────
│ val s2u   = 0
│ Reg Trust = True
│ base      = addr pad 32
└────────────────────────────
```

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

```
┌─ StackRelativeAddressing ──────────────────
│ DataInstruction
├────────────────────────────────────────────
│ val s2u = 1
│ base    = (addr pad 32) plus (Reg F pad 32)
└────────────────────────────────────────────
```

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.

```
┌─ ProgramCounterRelativeAddressing ─────────
│ DataInstruction
├────────────────────────────────────────────
│ val s2u = 2
│ base    = (addr pad 32) plus (Reg P pad 32)
└────────────────────────────────────────────
```

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).

```
┌─ AbsoluteAddressing ─
│ AddressBases
├──────────────────────
│ val s21 = 0
│ offs    = base
└──────────────────────
```

Absolute Addressing. The location to read in from is simply the base address defined above.

```
┌─ XIndexedAddressing ──────────
│ AddressBases
├───────────────────────────────
│ val s21 = 1
│ offs    = base plus (Reg X)
└───────────────────────────────
```

Indexed Addressing using the X index register. The location to read in 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.

```
┌─ YIndexedAddressing ──────────
│ AddressBases
├───────────────────────────────
│ val s21 = 2
│ offs    = base plus (Reg Y)
└───────────────────────────────
```

Indexed Addressing using the Y index register. The location to read in 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.

```
┌─ ZIndexedAddressing ──────────
│ AddressBases
├───────────────────────────────
│ val s21 = 3
│ offs    = base plus (Reg Z1)
└───────────────────────────────
```

Indexed Addressing using the Z index register. The location to read in 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.

```
IndexedAddressing ≙ ZIndexedAddressing ∨ YIndexedAddressing
                    ∨ AbsoluteAddressing ∨ XIndexedAddressing
```

All of the simple addressing modes.

```
┌─ IndexAddressing ─────────────
│ IndexedAddressing
├───────────────────────────────
│ val fc ≠ 13
│ val s2 < 12
│ m = Mem (offs trim 20)
│ io = 0
└───────────────────────────────
```

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 m 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.

```
┌─ ImmediateAddressing ──────────────────
│ DataInstruction
├────────────────────────────────────────
│ (val s2 = 12)∧(m = addr pad 32) ∨
│ (val s2 = 13)∧(m = wnot (addr 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 s2 is 12. If s2 is 13 then the m input is this value inverted (ie 1's complement). This allows both negative and positive values to be used as constants.

4.3 Access to General Purpose Registers

```
┌─ RegisterAddress ──
│ DataInstruction
├──────────────────
│ val s2 = 14
└──────────────────
```

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.

```
┌─ UseRegisterA ──────
│ RegisterAddress
├────────────────────
│ (val addr ) mod 4 = 0
│ m = Reg A
└────────────────────
```

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

```
┌─ UseRegisterX ──────
│ RegisterAddress
├────────────────────
│ (val addr ) mod 4 = 1
│ m = Reg X
└────────────────────
```

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

```
┌─ UseRegisterY ──────
│ RegisterAddress
├────────────────────
│ (val addr ) mod 4 = 2
│ m = Reg Y
└────────────────────
```

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

```
┌─ UseRegisterZ ──────
│ RegisterAddress
├────────────────────
│ (val addr ) mod 4 = 3
│ m = Reg Z1
└────────────────────
```

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

```
┌─StackNctSet ──────────
│ DataInstruction
│ ErrorInstruction
├──────────────────────
│ val s2u = 1
│ Reg NoStack = True
│ δMem = {}
└──────────────────────
```

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

```
┌─UnsetX ──────────
│ AddressBases
│ ErrorInstruction
├──────────────────
│ val s21 = 1
│ Reg IX = True
│ δMem = {}
└──────────────────
```

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

```
┌─UnsetY ──────────
│ AddressBases
│ ErrorInstruction
├──────────────────
│ val s21 = 2
│ Reg IY = True
│ δMem = {}
└──────────────────
```

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

```
┌─UnsetZ ──────────
│ AddressBases
│ ErrorInstruction
├──────────────────
│ val s21 = 3
│ Reg IZ = True
│ δMem = {}
└──────────────────
```

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

```
UnsetAddressingRegister ≙ StackNotSet
                          ∨ UnsetZ
                          ∨ UnsetY
                          ∨ UnsetX
```

The four cases of illegally used registers.

```
┌─ IllegalStackAddress ──────────────────────────
│ IndexedAddressing
│ ErrorInstruction
├────────────────────────────────────────────────
│ val fc ≠ 13
│ val s2u = 1
│ ((val offs) < (val (Reg F)) ∨
│  (val offs) > (val (Reg F)) + (val (Reg S)))
│ δMem = {}
└────────────────────────────────────────────────

┌─ IllegalReadAddress ─
│ IndexedAddressing
│ ErrorInstruction
├──────────────────────
│ val fc ≠ 13
│ val s2 < 12
│ invalid offs = 1
│ δMem = {}
└──────────────────────
```

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
│ δMem = {}
└──────────────────────
```

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
│ δMem = {}
└──────────────────────
```

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
│ δMem = {}
└──────────────────────
```

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
│ δMem = {}
└──────────────────────
```

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 m input to the ALU.

```
┌─RegisterSelectInvalidError ──────
│ ΔViper2
│ ErrorInstruction
├──────────────────────────────────
│ ( (val s2 ≠ 15 ∧ val fc ≠ 13 ) ∨
│   (val fq = 3  ∧ val fc < 12 ) ∨
│   (val fc = 6) )
│
│ δMem = {}
└──────────────────────────────────
```

The instruction selected requires a register be the r input to the ALU. (ie either a dyadic data instruction s2 ≠ 15 and fc ≠ 13, a write instruction s2 = 15, fq = 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.

```
┌─RegisterSelectAInvalid ─────
│ RegisterSelectInvalidError
├─────────────────────────────
│ val s1 = 0
│ Reg IA = True
└─────────────────────────────
```

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

```
┌─RegisterSelectXInvalid ─────
│ RegisterSelectInvalidError
├─────────────────────────────
│ val s1 = 1
│ Reg IX = True
└─────────────────────────────
```

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

```
┌─RegisterSelectYInvalid ─────
│ RegisterSelectInvalidError
├─────────────────────────────
│ val s1 = 2
│ Reg IY = True
└─────────────────────────────
```

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

```
┌─RegisterSelectZInvalid ─────
│ RegisterSelectInvalidError
├─────────────────────────────
│ val s1 = 3
│ Reg IZ = True
└─────────────────────────────
```

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

$$\begin{aligned} RegisterSelectInvalid \triangleq\ & RegisterSelectZInvalid \\ \vee\ & RegisterSelectYInvalid \\ \vee\ & RegisterSelectXInvalid \\ \vee\ & RegisterSelectAInvalid \end{aligned}$$

The four cases of illegal register being used for the r input to the ALU.

4.6 Comparison Operations

CompareFrame

RegisterSelect
MemoryRead
Bresult : $Word_1$

$\delta Mem = \{\}$

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

GreaterThanOrEqualTo

CompareFrame

---

val fc = 0
Bresult = wrd 1 (not (r less m))

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

EqualTo

CompareFrame

---

val fc = 1
Bresult = wrd 1 (r equal m)

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

GreaterThan

CompareFrame

---

val fc = 2
Bresult = wrd 1 (not((r less m) + (r equal m)))

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

UnsignedLessThan

CompareFrame

---

val fc = 3
Bresult = wrd 1 (r borrow m)

Bresult is set true if the r input, treated as an unsigned integer, is less than the m input.

```
┌─ AndEqualZero ──────────────────────────
│ CompareFrame
├─────────────────────────────────────────
│ val fc  = 4
│ Bresult = wrd 1 ((r and m) equal (zero))
└─────────────────────────────────────────
```

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

```
CompOp  ≙ AndEqualZero ∨ UnsignedLessThan ∨ GreaterThan
          ∨ EqualTo ∨ GreaterThanOrEqualTo
```

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

```
┌─ Condition ─────────────────────
│ CompOp
├─────────────────────────────────
│ val fq = 0
│ δReg   = {B↦Bresult,IB↦False}
└─────────────────────────────────
```

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

```
┌─ NotCondition ──────────────────────
│ CompOp
├─────────────────────────────────────
│ val fq = 1
│ δReg   = {B↦wnot(Bresult),IB↦False}
└─────────────────────────────────────
```

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

```
┌─ BorCondition ──────────────────────────
│ CompOp
├─────────────────────────────────────────
│ val fq = 2
│ δReg   = {B↦( Reg(B) or Bresult),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.

```
┌─ BorNotCondition ─────────────────────────────────
│ CompOp
├───────────────────────────────────────────────────
│ val fq = 3
│ δReg   = {B↦(Reg (B) or wnot(Bresult)),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 ≙ Condition ∨ BorNotCondition
          ∨ BorCondition ∨ NotCondition
```

The four operations loading B with a result. There are 15 × 4 × 20 = 1200 compare operations out of the possible 2^{12} Viper2 operations.

4.7 Viper2 Arithmetic

```
┌─ ALUInstruction ──────
│ RegisterSelect
│ MemoryRead
│ δB : RegName ⇸ Word
├──────────────────────
│ δMem = {}
└──────────────────────
```

Framing schema for all of the ALU operations. Note memory cannot be changed. δB holds any changes to the B register.

```
┌─ SignedAdd ───────
│ ALUInstruction
├──────────────────
│ val fc  = 5
│ Result  = r plus m
│ δB      = {}
└──────────────────
```

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

```
┌─ UnsignedAdd ─────────────────────────────
│ ALUInstruction
├──────────────────────────────────────────
│ val fc  = 6
│ Result  = r plus m
│ δB      = {B↦wrd 1 (r carry m),IB↦False}
└──────────────────────────────────────────
```

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

```
┌─ SignedSubtract ───
│ ALUInstruction
├───────────────────
│ val fc  = 7
│ Result  = r minus m
│ δB      = {}
└───────────────────
```

Subtract r from m. There is no check for underflow, this is done later in an error schema.

```
┌─ UnsignedSubtract ──────────────────────────
│ ALUInstruction
├─────────────────────────────────────────────
│ val fc  = 8
│ Result  = r minus m
│ δB      = {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
│ δB      = {}
└───────────────────────────
```

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

```
ArithmeticOp ≙ ( UnsignedAdd ∨ SignedSubtract ∨ SignedMultiply
                 ∨ UnsignedSubtract  ∨ SignedAdd )
```

The five arithmetic operations. There are 15 × 4 × 5 = 300 possible operations (ie 15 addressing modes by four register inputs by five possible operations).

4.8 Logical Operations

```
┌─ExclusiveOr ─────────
│ ALUInstruction
├────────────────────
│ val fc   = 11
│ Result   = r exor m
│ δB       = {}
└────────────────────
```

Returns the exclusive or of the two input words.

```
┌─And ─────────────────
│ ALUInstruction
├────────────────────
│ val fc   = 9
│ Result   = r and m
│ δB       = {}
└────────────────────
```

Returns the logical and of the two inputs.

```
┌─Or ──────────────────
│ ALUInstruction
├────────────────────
│ val fc   = 10
│ Result   =  r or m
│ δB       = {}
└────────────────────
```

Returns the logical or of the two inputs.

LogicalOp ≙ (Or ∨ And ∨ ExclusiveOr)

The three logical operators. There are 15 × 4 × 3 = 180 possible logical operations.

4.9 Load Instruction

```
┌─MonadicInstruction───
│ IndexedAddressing
│ δB : RegName ⇸ Word
├──────────────────
│ val fc = 13
└──────────────────
```

The operation is a monadic or load instruction. There is no register select, the only operand comes from the m input to the ALU. The register select field s1 is used to determine which operation is performed.

```
┌─LoadRegister─────────
│ MonadicInstruction
├──────────────────
│ val s1  = 0
│ Result  = Mem(offs trim 20)
│ δB      = {}
│ 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))
│ δB      = {}
│ 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
│ δB      = {}
│ val s2 ≤ 12
│ io      = 0
└──────────────────
```

Load the address detemined by the addressing mode into the result.

```
┌─ InputFromPERI ──────────────────
│ MonadicInstruction
├──────────────────────────────────
│ val s1  = 3
│ Result  = Mem(offs trim 20)
│ δB      = {}
│ val s2 ≤ 3
│ io      = 1
└──────────────────────────────────
```

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.

4.10 Destination Registers

```
┌─ ResultToA ──────────────────────────
│ ALU
├──────────────────────────────────────
│ val fq = 0
│ δReg   = {A↦Result,IA↦False} ⊕ δB
└──────────────────────────────────────
```

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.

```
┌─ ResultToX ──────────────────────────
│ ALU
├──────────────────────────────────────
│ val fq = 1
│ δReg   = {X↦Result,IX↦False} ⊕ δB
└──────────────────────────────────────
```

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.

```
┌─ ResultToY ──────────────────────────
│ ALU
├──────────────────────────────────────
│ val fq = 2
│ δReg   = {Y↦Result,IY↦False} ⊕ δB
└──────────────────────────────────────
```

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.

```
┌─ ResultToZ1 ─────────────────────────
│ ALU
├──────────────────────────────────────
│ val fq = 3
│ δReg   = {Z1↦Result,IZ↦False} ⊕ δB
└──────────────────────────────────────
```

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.

ALUOp ≙ ResultToA ∨ ResultToX ∨ ResultToY ∨ ResultToZ1

Load one of the four general purpose registers. There are 540 × 4 = 2160 possable operations. The two other function codes fc = 13, fc = 14 will give annother 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
├──────────────────
│ val fc  = 5
│ (r overflow m) = 1
│ δMem = {}
└──────────────────
```

An overflow has occured on a signed add.

```
┌─ SignedSubtractUndeflow ──
│ ALUInstruction
│ ErrorInstruction
├──────────────────
│ val fc  = 7
│ (r underflow m) = 1
│ δMem = {}
└──────────────────
```

An underflow has occured on a signed subtract.

```
┌─ SignedMultiplyOverflow ──
│ ALUInstruction
│ ErrorInstruction
├──────────────────
│ val fc  = 12
│ (r overflow m) = 1
│ δMem = {}
└──────────────────
```

An overflow has occured on a signed multiply.

```
┌─ LoadAndNegateRegisterOverflow ──
│ MonadicInstruction
│ ErrorInstruction
├──────────────────
│ val s1  = 1
│ m = Mem(offs trim 20)
│ (zero underflow m) = 1
│ δMem = {}
└──────────────────
```

An underflow has occured when loading and negating a register. This means that the value which was loaded must have been maxneg.

```
┌─ LoadEffectiveAddressError ──
│ MonadicInstruction
│ ErrorInstruction
├──────────────────────
│ val s1  = 2
│ val s2 > 12
│ δMem = {}
└──────────────────────
```

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

```
┌─ InputFromPERIError ──
│ MonadicInstruction
│ ErrorInstruction
├──────────────────
│ val s1  = 3
│ val s2 > 3
│ δMem = {}
└──────────────────
```

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

```
┌─ IllegalAddress ──────
│ MonadicInstruction
│ ErrorInstruction
├──────────────────
│ val s1 ≠ 2
│ invalid offs = 1
│ δMem = {}
└──────────────────
```

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

```
MonError ≙ LoadEffectiveAddressError ∨
           InputFromPERIError

MonadError ≙ MonError ∨
             ¬ (MonError) ∧ IllegalAddress

MonadicError ≙ MonadError ∨
               ¬ (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 ∨
             SignedSubtractUndeflow ∨
             SignedMultiplyOverflow ∨
             MonadicError
```

The Errors which can occur during ALU operations.

4.12 Jumps and Calls

```
┌─ControlInstruction ─
│ MemoryRead
├────────────────
│ val s2 = 15
│ val fq ≠ 3
└────────────────
```

The instruction is a control instruction.

```
┌─DestinationSelect ──────────────────────────
│ ControlInstruction
│ Destination : Word32
├─────────────────────────────────────────
│ (val fq = 0)∧(Destination = addr pad 32)
│ ∨
│ (val fq = 1)∧
│ (Destination = (addr pad 32) plus  (newp pad 32))
│ ∨
│ (val fq = 2)∧
│ (Destination = (addr pad 32) minus (newp pad 32))
└─────────────────────────────────────────
```

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.

```
┌─UnconditionalJump ─────────────
│ DestinationSelect
├──────────────────────────
│ val fc = 0
│ δReg = {P↦(Destination trim 20)}
│ δMem   = {}
└──────────────────────────
```

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

```
┌─JumpIfError ─────────────────────────
│ DestinationSelect
├───────────────────────────────────
│ val fc = 1
│ Reg Trust = True
│ val (Reg E) = 1
│ δReg = {P↦(Destination trim 20),IA↦False,
│           IX↦False,IY↦False,IZ↦False,IB↦False}
│ δMem   = {}
└───────────────────────────────────
```

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

```
JumpIfBSet
DestinationSelect
------------------------------
val fc = 2
val (Reg B) = 1
δReg = {P↦(Destination trim 20)}
δMem   = {}
```

Jump if the B flag is set.

```
JumpIfBNotSet
DestinationSelect
------------------------------
val fc = 3
val (Reg B) = 0
δReg = {P↦(Destination trim 20)}
δMem   = {}
```

Jump if the B flag is not set.

```
JumpIfAttentionSet
DestinationSelect
------------------------------
val fc = 4
attention = 1
δReg = {P↦(Destination trim 20)}
δMem   = {}
```

Jump if the attention input to the Viper2 microprocessor is set.

```
JumpIfAttentionNotSet
DestinationSelect
------------------------------
val fc = 5
attention = 0
δReg = {P↦(Destination trim 20)}
δMem   = {}
```

Jump if the attention input to the Viper2 microprocessor is not set.

```
FailedJumpCondition
ControlInstruction
ΞViper2
------------------------------
((val fc = 1) ∧ (val (Reg E) = 0))  ∨
((val fc = 2) ∧ (val (Reg B) = 0))  ∨
((val fc = 3) ∧ (val (Reg B) = 1))  ∨
((val fc = 4) ∧ (attention = 0))    ∨
((val fc = 5) ∧ (attention = 1))
```

If the Jump condition is false, then Viper2 state the same (apart from the Program counter increment).

```
DecrementAndJumpOnNotZero ______________________________
| DestinationSelect
| RegisterSelect
| δPC    : RegName ⇸ Data
|_____________________________________________________
| val fc = 6
| Result = r minus one
| (Result ≠ zero) ⇒ (δPC = {P↦(Destination trim 20)})
| (Result = zero) ⇒ (δPC = {})
| (val s1 = 0) ⇒ (δReg = {A↦Result,IA↦False}  ⊕ δPC)
| (val s1 = 1) ⇒ (δReg = {X↦Result,IX↦False}  ⊕ δPC)
| (val s1 = 2) ⇒ (δReg = {Y↦Result,IY↦False}  ⊕ δPC)
| (val s1 = 3) ⇒ (δReg = {Z1↦Result,IZ↦False} ⊕ δPC)
| δMem   = {}
|_____________________________________________________
```

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

```
CallInstruction ______________________________________________
| DestinationSelect
| δFlags : RegName ⇸ Data
| TopOfCallFrame,BottomOfNewWorkspace : Data
| BottomOfCallFrame,ProgramStatusWord : Data
|____________________________________________________________
| BottomOfCallFrame     = ((Reg F) pad 32) plus
|                                          ((Reg S) pad 32)
| TopOfCallFrame        = BottomOfCallFrame  plus  one
| ProgramStatusWord     = (Reg P) pad 32 ⊕ {20↦val(Reg Trust)}
| BottomOfNewWorkspace = TopOfCallFrame  plus  one
|
| δMem        = {(BottomOfCallFrame trim 20)↦(Reg F),
|                (TopOfCallFrame trim 20)↦(ProgramStatusWord)}
| δReg        = {F↦(BottomOfNewWorkspace),
|                P↦(Destination trim 20),Postcall↦True}
|               ⊕ 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 __
| CallInstruction
|___________________
| val fc = 8
| δFlags = {}
|___________________
```

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

```
┌─CallIfError ──────────────────────────
│ CallInstruction
├──────────────────────────────────────
│ val fc = 9
│ val (Reg E) = 1
│ δFlags = {IA↦False,IX↦False,IY↦False,
│           IZ↦False,IB↦False,E↦False}
└──────────────────────────────────────
```

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

```
┌─CallIfBSet ─────────
│ CallInstruction
├─────────────────────
│ val fc = 10
│ val (Reg B) = 1
│ δFlags = {}
└─────────────────────
```

Call if the B flag is set.

```
┌─CallIfBNotSet ──────
│ CallInstruction
├─────────────────────
│ val fc = 11
│ val (Reg B) = 0
│ δFlags = {}
└─────────────────────
```

Call if the B flag is not set.

```
┌─CallIfAttentionSet ─
│ CallInstruction
├─────────────────────
│ val fc = 12
│ attention = 1
│ δFlags = {}
└─────────────────────
```

Call if the attention input to the Viper2 microprocessor is set.

```
┌─CallIfAttentionNotSet ─
│ CallInstruction
├────────────────────────
│ val fc = 13
│ attention = 0
│ δFlags = {}
└────────────────────────
```

Call if the attention input to the Viper2 microprocessor is not set.

```
┌─ FailedCallCondition ──────────────────────────
│ ControlInstruction
│ ΞViper2
├────────────────────────────────────────────
│ ((val fc = 9)  ∧ (val (Reg E) = 0))  ∨
│ ((val fc = 10) ∧ (val (Reg B) = 0))  ∨
│ ((val fc = 11) ∧ (val (Reg B) = 1))  ∨
│ ((val fc = 12) ∧ (attention = 0))    ∨
│ ((val fc = 13) ∧ (attention = 1))
└────────────────────────────────────────────
```

If the Call condition is false, then Viper2 state the same (apart from the Program counter increment).

4.13 Copy Instruction

```
┌─ CopyFromRegisterToGeneralPurposeRegister ─────
│ ControlInstruction
│ ad : N
├────────────────────────────────────────────
│ val fc = 7
│ val fq = 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 S ) pad 32 )
│ (ad = 7 )  ⇒ (Result = (Reg U ) pad 32 )
│ (ad = 8 )  ⇒ (Result = (Reg Watchdog ) pad 32 )
│ (ad = 9 )  ⇒ (Result = Reg D )
└────────────────────────────────────────────
```

Copy from a register to a gebneral purpose register.

```
┌─ CopyToGeneralPurposeRegister ─────────
│ CopyFromRegisterToGeneralPurposeRegister
├────────────────────────────────────────────
│ (val s1 = 0 ) ⇒ (δReg = {A↦Result,IA↦False} )
│ (val s1 = 1 ) ⇒ (δReg = {X↦Result,IX↦False} )
│ (val s1 = 2 ) ⇒ (δReg = {Y↦Result,IY↦False} )
│ (val s1 = 3 ) ⇒ (δReg = {Z1↦Result,IZ↦False} )
└────────────────────────────────────────────
```

Place value in general purpose register.

```
┌─ CopyFromGeneralPurposeRegisterToRegister ─────────────
│ ControlInstruction
│ RegisterSelect
│ ad : N
├────────────────────────────────────────────────────
│ val fc = 7
│ val fq = 1
│ Reg Trust = True
│ ad      = (val addr ) mod 16
│ (ad = 0 )  ⇒ (δReg = {A↦r ,IA↦False} )
│ (ad = 1 )  ⇒ (δReg = {X↦r ,IX↦False} )
│ (ad = 2 )  ⇒ (δReg = {Y↦r ,IY↦False} )
│ (ad = 3 )  ⇒ (δReg = {Z1↦r ,IZ↦False} )
│ (ad = 4 )  ⇒ (δReg = {P↦(r trim 20 )} )
│ (ad = 5 )  ⇒ (δReg = {F↦(r trim 20 ),NoLimit↦True,
│                          NoSize↦True,NoStack↦False} )
│ (ad = 6 )  ⇒ (δReg = {S↦(r trim 20 ),NoSize↦False} )
│ (ad = 7 )  ⇒ (δReg = {U↦(r trim 20 ),NoLimit↦False} )
│ (ad = 8 )  ⇒ (δReg = {Watchdog↦(r trim 16 ),WE↦False} )
│ (ad = 9 )  ⇒ (δReg = {D↦r} )
└────────────────────────────────────────────────────
```

Copy a value from a general purpose register to a special register.

4.14 Enter and Return

```
┌─Enter ──────────────────────────────────────────────────────
│ ControlInstruction
├─────────────────────────────────────────────────────────────
│ val fc = 14
│ (val (Reg F)) + (val addr) + 2 ≤ (val (Reg U))
│ Reg Postcall  = True
│ (val fq = 0) ⇒ (δReg ={S↦addr,Postcall↦False})
│ (val fq = 1) ⇒ (δReg ={S↦addr,Trust↦False,Postcall↦False})
│ (val fq = 2) ⇒ (δReg ={S↦addr,Trust↦True,Postcall↦False})
└─────────────────────────────────────────────────────────────
```

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 accomadate a call instruction in the new routine. It also sets up the trusted ness of the routine. Finally the postcall bit is reset.

```
┌─Return ─────────────────────────────────────────────
│ CallInstruction
├─────────────────────────────────────────────────────
│ TopOfCallFrame         = (Reg F) minus one
│ ProgramStatusWord      = Mem(TopOfCallFrame)
│ BottomOfCallFrame      = TopOfCallFrame minus one
│ BottomOfNewWorkspace = Mem(BottomOfCallFrame)
│ δReg        = {F↦(BottomOfNewWorkspace),
│                P↦(ProgramStatusWord trim 20),
│                Trust↦(ProgramStatusWord 20),
│                S↦(BottomOfCallFrame minus
│                   (BottomOfNewWorkspace trim 20))}
└─────────────────────────────────────────────────────
```

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 ≙   CopyFromGeneralPurposeRegisterToRegister  ∨
           CopyToGeneralPurposeRegister
```

The two copy commands. This covers 1 × 4 × 1 × 2 = 8 operations.

```
Jump  ≙  UnconditionalJump  ∨ JumpIfError ∨
         JumpIfAttentionSet ∨ JumpIfBNotSet ∨
         DecrementAndJumpOnNotZero ∨ JumpIfAttentionNotSet ∨
         JumpIfBSet


Jumps  ≙   Jump ∨ FailedJumpCondition
```

The seven jump commands. This covers 1 × 4 × 7 × 3 = 84 operations.

```
Call  ≜  UnconditionalCall ∨ CallIfError ∨
          CallIfBSet ∨ CallIfBNotSet ∨ CallIfAttentionSet ∨
          CallIfAttentionNotSet

Calls  ≜  Call ∨ FailedCallCondition
```

The six call commands. This covers 1 × 4 × 6 × 3 = 72 operations.

```
Control ≜ Calls ∨ Jumps ∨ Copies ∨ Enter ∨ Return
```

The control operations. There are 8 + 84 + 72 + 1 × 4 × 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) ∨ (val fc = 3)
└─────────────────────────────
```

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

```
IllegalJumps  ≙ IllegalJump  ∨
                IllegalJumpCondition
```

```
┌─ IllegalCallError ─────────────────────────────────────────────
│ ControlInstruction
│ ErrorInstruction
│ DestinationSelect
│ TopOfCallFrame,BottomOfNewWorkspace : Data
│ BottomOfCallFrame,ProgramStatusWord : Data
├────────────────────────────────────────────────────────────────
│ (val fc ≥ 8) ∧ (val fc ≤ 13)
│ BottomOfCallFrame     = ((Reg F) pad 32) plus
│                                             ((Reg S) pad 32)
│ TopOfCallFrame        = BottomOfCallFrame  plus  one
│ ProgramStatusWord     = (Reg P) pad 32 ⊕ {20↦val(Reg Trust)}
│ BottomOfNewWorkspace = TopOfCallFrame  plus  one
└────────────────────────────────────────────────────────────────
```

Framing schema for Call errors.

```
┌─ IllegalDestination ──────
│ IllegalCallError
├───────────────────────────
│ invalid Destination = 1
│ δMem     = {}
└───────────────────────────
```

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

```
┌─ IllegalBottomOfCallFrame ────────
│ IllegalCallError
├──────────────────────────────────
│ invalid BottomOfCallFrame = 1
│ δMem    = {}
└──────────────────────────────────
```

The bottom of the call space is not in memory.

```
┌─ IllegalTopOfCallFrame ─────────────────────────
│ IllegalCallError
├─────────────────────────────────────────────────
│ invalid TopOfCallFrame = 1
│ δMem        = {(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
│ δMem        = {(BottomOfCallFrame trim 20)↦(Reg F),
│                (TopOfCallFrame trim 20)↦(ProgramStatusWord)}
└─────────────────────────────────────────────────────────
```

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

```
┌─ StackNotSet ──────────────────────────────
│ IllegalCallError
├────────────────────────────────────────────
│ (Reg NoStack = True) ∨ (Reg NoSize = True)
│ δMem    = {}
└────────────────────────────────────────────
```

A call has been made with the stack not set.

```
IllegalCalls ≙ IllegalDestination          ∨
               IllegalBottomOfCallFrame    ∨
               IllegalTopOfCallFrame       ∨
               IllegalBottomOfNewWorkspace ∨
               StackNotSet
```

All of the Illegal Call schemas.

4.15 Illegal Copy

```
┌─ CopyError1 ──────────────────────────────────
│ ControlInstruction
│ ErrorInstruction
├───────────────────────────────────────────────
│ val fc = 7
│ val fq = 0
│ δMem   = {}
│ ((val addr = 0) ∧ (Reg IA = True)       ∨
│  (val addr = 1) ∧ (Reg IX = True)       ∨
│  (val addr = 2) ∧ (Reg IY = True)       ∨
│  (val addr = 3) ∧ (Reg IZ = True)       ∨
│  (val addr = 5) ∧ (Reg NoStack = True)  ∨
│  (val addr = 6) ∧ (Reg NoSize = True)   ∨
│  (val addr = 7) ∧ (Reg NoLimit = True)  )
└───────────────────────────────────────────────
```

Attempt to copy invalid register.

```
┌─ CopyError2 ─────────────────────────
│ ControlInstruction
│ ErrorInstruction
├──────────────────────────────────────
│ val fc = 7
│ val fq = 1
│ Reg Trust = True
│ δMem   = {}
│ ((val s1 = 0) ∧ (Reg IA = True) ∨
│  (val s1 = 1) ∧ (Reg IX = True) ∨
│  (val s1 = 2) ∧ (Reg IY = True) ∨
│  (val s1 = 3) ∧ (Reg IZ = True) )
└──────────────────────────────────────
```

Attempt to copy invalid register.

```
┌─ IllegalCopy ──────────
│ ControlInstruction
│ ErrorInstruction
├────────────────────────
│ val fc = 7
│ val fq = 1
│ Reg Trust = False
│ δMem   = {}
└────────────────────────
```

Attempt to copy to protected register, in untrusted mode.

    IllegalCopies ≙ CopyError1 ∨ CopyError2 ∨ IllegalCopy

Error in copying from register to register.

```
┌─LimitNotSet ──────────
│ ControlInstruction
│ ErrorInstruction
├──────────────────────
│ val fc = 14
│
│ δMem    = {}
└──────────────────────
```

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, ie Enter has occured 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, ie Enter has not occured 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 accomadate the present frame.

4.17 Write Operations

```
 WriteInstruction
 RegisterSelect
 ─────────────
 val s2 = 15
 val fq = 3
 val fc ≤ 11
 δReg   = {}
```

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

```
 GlobalWrite
 WriteInstruction
 ─────────────
 val fch = 0
 base     = addr pad 32
 io       = 0
 Reg Trust = True
```

Write to Global memory.

```
 LocalStackFrameWrite
 WriteInstruction
 ─────────────
 val fch = 1
 base     = (addr pad 32) plus (Reg F)
 io       = 0
```

Write to local stack frame.

```
 OutputToPERI
 WriteInstruction
 ─────────────
 val fch = 2
 base     = addr pad 32
 io       = 1
```

Output to PERIpheral.

```
    WriteBase ≙ GlobalWrite ∨ LocalStackFrameWrite ∨ OutputToPERI
```

The three addressing modes.

```
 Write
 WriteBase
 ─────────────
 (val fcl = 0 ∧ offs = base                  ∨
  val fcl = 1 ∧ offs = base plus (Reg X)     ∨
  val fcl = 2 ∧ offs = base plus (Reg Y)     ∨
  val fcl = 3 ∧ offs = base plus (Reg 21) )
 δMem = {(offs trim 20) ↦ r}
```

Write to the location specified. Either absolute addressing or indexed addressing. Write has 1 × 4 × 1 × 12 = 48 Operations.

4.18 Write Errors

```
┌─WriteError ──────────────────────────────────────
│ WriteInstruction
│ ErrorInstruction
├──────────────────────────────────────────────────
│ (val fcl = 0) ∧ (offs = base)                   ∨
│ (val fcl = 1) ∧ (offs = base plus (Reg X)) ∨
│ (val fcl = 2) ∧ (offs = base plus (Reg Y)) ∨
│ (val fcl = 3) ∧ (offs = base plus (Reg Z1))
└──────────────────────────────────────────────────
```

Write error framing schema.

```
┌─IllegalIndex ───────────────────────
│ WriteInstruction
│ ErrorInstruction
├─────────────────────────────────────
│ (val fcl = 1) ∧ (Reg IX = True) ∨
│ (val fcl = 2) ∧ (Reg IY = True) ∨
│ (val fcl = 3) ∧ (Reg IZ = True)
└─────────────────────────────────────
```

The index register specified is illegal.

```
┌─GlobalWriteError ─
│ WriteError
├───────────────────
│ val fch = 0
│ δMem = {}
│ invalid offs = 1
└───────────────────
```

The write location is not in the memory space.

```
┌─StackFrameWriteError ─────────────────────────────
│ WriteError
├───────────────────────────────────────────────────
│ val fch = 1
│ δMem = {}
│ ( invalid offs = 1                               ∨
│  (val offs) < (val (Reg F))                      ∨
│  (val offs) > (val (Reg F)) + (val (Reg S)) )
└───────────────────────────────────────────────────
```

The write location is not in the stack frame.

```
┌─GlobalOutputError ─
│ WriteError
├────────────────────
│ val fch = 2
│ δMem = {}
│ invalid offs = 1
└────────────────────
```

The output location is not in the memory space.

```
IllegalWriteAddress ≙ GlobalWriteError      ∨
                      StackFrameWriteError ∨
                      GlobalOutputError     ∨
                      IllegalIndex
```

The Write Errors.

4.19 Other Viper2 Errors

```
┌─ WatchdogTimout ──────
│ ΔViper2
│ ErrorInstruction
├───────────────────────
│ Reg WE    = True
│ Reg Trust = False
└───────────────────────
```

The watchdog timer has timed out, and Viper2 is in untrusted mode.

```
┌─ IllegalOpCode ─────────────────
│ ΔViper2
│ ErrorInstruction
├─────────────────────────────────
│ val s2 = 15
│ ((val fc = 6) ∧ (val fq = 2)) ∨
│ ((val fq = 3) ∧ (val fc ≥12))
└─────────────────────────────────
```

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

```
Viper2_Error ≙ IllegalP                     ∨
               UnsetAddressingRegister      ∨
               RegisterSelectInvalid        ∨
               RegisterInvalid              ∨
               IllegalReadAddress           ∨
               ArithError                   ∨
               IllegalJumps                 ∨
               IllegalCopies                ∨
               LimitNotSet                  ∨
               EnterNotFound                ∨
               StackOverflow                ∨
               PostcallNotSet               ∨
               IllegalCalls                 ∨
               IllegalWriteAddress          ∨
               WatchdogTimout               ∨
               IllegalOpCode
```

The Viper2 Error conditions.

```
│ arb : Word → Word
├─────────────────────────────────────
│ ∀ w1,w2 : Word | #w1 = #w2 • w1 = arb w2
```

The arb function, ie no relationship between input and output words

```
┌─ TrustedError ──────
│ Viper2_Error
├─────────────────────
│ Reg Trust = True
│ stop' = 1
└─────────────────────
```

Error in trusted state, machine stops.

```
UntrustedError
  Viper2_Error
  CallInstruction
  ______________________________________________
  Reg Trust ≡ False
  TopOfCallFrame          = (Reg F) minus one
  ProgramStatusWord       = Mem(TopOfCallFrame)
  BottomOfCallFrame       = TopOfCallFrame minus one
  BottomOfNewWorkspace = Mem(BottomOfCallFrame)
  δReg = {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↦(BottomOfNewWorkspace),
          P↦(ProgramStatusWord trim 20),
          Trust↦(ProgramStatusWord 20),
          S↦(BottomOfCallFrame minus
                      (BottomOfNewWorkspace trim 20))}
  stop' = 0
```

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

Viper2_Errors ≙ UntrustedError ∨ TrustedError

4.20 The Viper Top Level Specification

```
┌─NotStopped──
│ ΔViper2
├────────────
│ stop' = 0
└────────────
```

ViperOK ≙ Compare ∨ ALUOp ∨ Control ∨ Write

Viper2 has successfully completed an operation. There are 1200 + 2640 + 188 + 48 + 20 = 4096 possible operations, ie all Op codes accounted for.

OKState ≙ ¬(Viper2_Errors) ∧ ViperOK ∧ NotStopped

NextState ≙ Viper2_Errors ∨ OKState ∨ Stopped ∨ 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 HOL specification.

C. Pygott and J. Kershaw for the design of the Viper2.

C. O'Halloran for his help with the Z editor and type checker.

S. 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.
3. Bowen J.	The Formal Specification of a Microprocessor Instruction Set.
4. Hayes I. (editor)	Specification Case Studies Prentice-Hall International series in computer science, 1987.

DOCUMENT CONTROL SHEET

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 box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memo 4217	3. Agency Reference	4. Report Security Classification U/C

5. Originator's Code (if known) 7784000	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN, WORCESTERSHIRE WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

Specification of Viper 2 in Z.

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials Kemp D H	9(a) Author 2	9(b) Authors 3,4...	10. Date 1988.10	pp. ref. 64

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

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.

S80/48