VP Reference Manual

Version 1.14

1. Introduction

VPcode is a virtual binary or portable byte code for a virtual machine. It is designed for translation into native code as efficiently as possible.

VPcode was designed with a small number of basic operations many of which can be combined into expression trees, thereby providing a powerful programming tool permitting compact binaries and also providing a simple mechanism for optimisation.

Virtual binaries are often considered slow when compared with native code, since historically virtual binaries were often interpreted. VPcode is translated to native code, which is analagous to the code generation process of the back end of a traditional compiler, giving comparable performance.

A major design feature of VPcode is that translation from the virtual binary into efficient native code occurs only when loaded from store (e.g. disk, network). This permits the translator to know exactly which processor the code will be running on and generate the appropriate code. For example, it may generate different code on an Intel 386 to that on an Intel Pentium, either because one contains instructions the other lacks, or because optimisation considerations differ between these processors. Therefore it is possible to achieve over 100% efficiency when compared with compiling or hand coding for a generic family.

2. Virtual Processor Model

2.1 Register banks

There are 5 register banks. Each register bank contains a specific type of data:

i	integer	32 bit
l	long	64 bit
f	float	32 bit
d	double	64 bit
p	pointer	32 bit

Each tool or subroutine gets its own local set of five banks of registers. Except for explicitly passed parameters, a subroutine's registers are not visible to other subroutines.

When making a subroutine call, any registers in each register bank can be used to pass actual parameters. In the subroutine, the formal parameters appear in register numbers starting at 0 in the respective bank.

2.1.1 Integer Registers

Integer registers may contain integers, bytes, characters etc. Integer registers start at i0.

All VPcode implementations support 32 bit integers.

2.1.2 Pointer Registers

In addition to the general purpose pointer registers there are also four special pointer registers:

sp	stack pointer
pp	parameter pointer
lp	link pointer (return jump address)
gp	global pointer

Pointer registers, like integer registers, are nominally 32 bits. On machines with larger pointer registers these larger registers may be used for pointers, but only 32 bits will be saved into memory or transferred to integer registers when a store to memory (cpy, st) or pointer to integer conversion (p2i) is performed.

All VPcode implementations support 32 bit pointers.

2.2 Special Registers

Special registers can be used in any place ordinary registers can be used, but any modifications made have to be carefully considered.

```
31                    0
   si   Signature Register
31                    0
   sp   Stack Pointer
   pp   Parameter Pointer
   lp   Link Pointer
   gp   Global Pointer
```

2.2.1 sp - stack pointer

The stack pointer points to the lowest address containing valid data. It should be noted that the stack is downward growing. The stack is always aligned to a CPU-specific alignment and this alignment is automatically preserved whenever it is adjusted.

Adjustment usually takes place with routine entry (*ent*, *entd*, *entl, entih*), return from routine (*ret*) or *als* instructions that explicitly allocate stack. See below for more information concerning the *als* instruction.

als instructions should not be placed in loops or other locations where they may be executed more than once (or not at all) as references can be made from the stack pointer using static offsets. Only use *als* to move the stack pointer.

The stack pointer should not be altered by *cpy.p x, sp*

2.2.2 pp - parameter pointer

The parameter pointer is a special read only register that is set to the stack pointer of the caller just before the call occurred. It is typically used for passing parameters on the stack. The standard Elate® parameter passing convention uses registers, except when there are variable arguments such as in *printf*.

pp is invalid in an *entd* block and in any subroutine with a *schk* instruction.

2.2.3 lp - link pointer

The link pointer contains the return address that the *ret* instruction will jump to after tidying up the stack in a return from routine.

lp should not be altered.

2.2.4 gp - global pointer

gp points to an area of thread-wide memory, which can be both global and static data. The size of the user definable area of this is specified in the main tool. (See Chapter 5.5.1 of *VP Tool Programming Guide*). The global pointer should not be altered by the application programmer.

2.3 The zap flag

The zap flag ~ (tilde) after a register name is used to indicate that the register is no longer needed.

If a subroutine or tool has return arguments that are not required in the calling tool, these can be discarded on the *gos* or *qcall* line using a zapped argument.

```
qcall lib/fputc,(i0 p2 : i~)
```

In VPcode it is illegal to put a ~ (tilde) after a register if it appears twice in the same expression.

In this case it is i1 that cannot be zapped.

```
cpy(i1 + (i3 * i1)),i4
```

If it is intended to zap a register that occurs more than once in an instruction, an explicit zap instruction should be added directly afterwards.

It is illegal to zap any of the special registers (*si, sp, lp, pp, gp*).

2.4 Block Structured Register Naming Macros

Registers can be renamed easily in the assembler. A set of definition (*def*) macros are available that allow names to be given to registers in a block structured fashion. These are held in *lang/asm/include/struct.inc* and are therefore normally included by default.

```
ent -:-
defbegin 0 ;checks that it is at the outer level
defp argv ;this will be p0
defi argc,val1,val2,cnt ;these are i0,i1,i2,i3
qcall lib/argcargv,(-:argv argc)
...
defend 0
```

The same register may have different names in different blocks and the same name may refer to a different register in different blocks.

If the *def* macros are used in a block then all the registers should be named using the *def* macros. A register name may be specified to say it is in use in a block - it must be specified in the same place as it would have been assigned to a name ordinarily.

```
defi abc,i1,i2,efg
```

abc equates to i0 and *efg* to i3.

It is not necessary to start the names with any particular letter.

The registers are zapped at the end of a block except for the outermost block as there is no point placing zaps after a *ret*. The outermost (or top level) block is one that is not enclosed in any other definition block.

There is a facility to suppress zaps at the end of a definition block, *defendnz*. This can also be used if desired just after a *go* instruction where the zaps are unnecessary.

The *defzap* macro puts in zaps without ending a block and can be used before a *go* instruction in an outer block. The utility 'dfa' will find most of the opportunities for zapping but the explicit zaps will allow a register used without a definition to be caught more often.

The following macros are provided:

defbegin	Start a block
defend	End a block and zap the registers freed
defendnz	End a block without any zaps
defzap	Zap the registers for the current block
defi	Define names for integer registers
defl	Define names for long registers
deff	Define names for float registers
defd	Define names for double registers
defp	Define names for pointer registers
defset	Equate a previously defined name to a register or number in a block

A detailed description is given under the entry for each macro.

2.5 Format of instructions

Instruction parameters can take various forms - constant, register, or expression. With these three simple forms, much more complex operations can be generated because expression forms may take expressions themselves.

Most VPcode operations are performed by expressions that are copied using the *cpy* instruction to a destination register.

Simple formats using constant or register would be:

```
cpy 6,i0
cpy i1,i2
```

These copy from the left to the destination register on the right.

Using an expression on the left, much more complicated operations can be performed:

```
cpy (i1 add 6),i3
```

Expressions themselves may take other expressions within them:

```
cpy (i1 add (i2 mul 6)),i3
```

The assembler will allow "+" for *add*, "*" for *mul*, "/" for *div* and "-" for *sub*.

Note the round brackets around the expressions.

```
cpy (1-symbol),i1 ;simple copy constant
cpy (1-i2),i1 ;subtract at run time
```

Macros are provided to do a *cpy* and an operation in one macro, so that the following are identical:

```
cpy (i0 add 6),i0
add 6,i0
```

The second is merely a macro generating the first line.

Note that the parameter order is reversed in the macro form. This is important if using *sub* or *div*!

```
cpy (i0-1),i0 ;subtracts 1 from i0
sub 1,i0 ;also subtracts 1 from i0
```

2.6 Tool structure

The code of a tool consists of a number of *ent blocks*, plus certain areas that are defined to be outside any *ent block*. Each *ent block* is a self contained subroutine and must have an entry directive as the first instruction. The standard entry instruction is *ent*.

```
      ent p0 i0-i2: - ;start of ent block
      ...
      gos routine2,(p3:i0)
      ...
      ret

routine2: ;tag
```

```
      ent p0:i0 ;start of next ent block
      ...
      ret
```

A further three entry instructions are available and are only to be used in the special circumstances outlined below:

entd	used at the start of a default method in object programming
entl	used for leaf tools and subroutines (ones that call no other tools or subroutines)
entih	for interrupt handlers only

Parameters passed in and out must be specified on the entry instruction. A single '-' denotes no parameters.

```
ent -:-     ;no input/output parameters
```

Certain rules apply to the instructions:

1. Outside any *ent block*, the only legal instructions are
   - tag - generated by defining a label
   - go tag
   - bcn (si ne constant),tag - generated by the method macro.

2. Any *go, bcn* or *bcp* instruction within an *ent block* cannot jump to a tag outside that *ent block*.

3. A gos instruction (that can only occur within an ent block) must call a tag that is outside any ent block. Using the gos register or gos expression variants can allow you to call a subroutine somewhere in a different tool. The same rule applies here - the target must be outside any ent block in that tool.

Behaviour is undefined if execution "falls off" the end of an *ent block* into the following *ent block* or area outside any *ent block*.

2.7 Calling mechanisms

VPcode provides three ways to call a subroutine: *gos*, *qcall* and method calls (*ncall/ccall/pcall*).

The call instruction has a list of input and output parameters. The called subroutine has its own set of registers, and on entry the input parameter registers in the calling routine (actual parameters) are copied to the subroutine's input parameter registers specified in the *ent* instruction (formal parameters). For each bank, the formal parameter registers are contiguous starting at register 0. On return from the subroutine, the output normal parameters specified on the *ent* line are copied to the output actual parameters specified on the call instruction. In both the input and output parameter lists, the numbers of each bank of registers must be matched between the call instruction and the *ent* line.

Example:

Caller:

```
gos aroutine,(i3 i5 f3 p2:l1 d4)    ;actual parameters
```

Subroutine:

```
aroutine:
      ent i0 i1 f0 p0:l0 d0 ;formal parameters
      ...
      ret
```

On entry, the caller's *i3,i5,f3,p2* are copied to the subroutine's *i0,i1,f0,p0*. On exit, the subroutine's *l0,d0* are copied to the caller's *l1,d4*.

2.7.1 gos

gos is used to call a subroutine in the same tool. The subroutine is identified by the name of the tag just before the *ent* line. Example as above.

gos can also be used with a pointer register or expression argument to call a subroutine whose address is given by the register or expression. This subroutine does not have to be in the same tool. The two typical ways of using this are to load the address of a subroutine into a pointer register directly for use by another tool or to use a lookup table.

Example 1

```
;subroutine address in pointer register for direct use by another tool
tool 'tool1'
      ...
      ent ...
      ...
      cpy.p aroutine,p0 ;load address of subroutine into p0
      qcall tool2,(p0:-)
      ...
aroutine: ;subroutine to call from 'tool2'
      ent i0 i1 f0 p0:l0 d0
      ...
      ret
toolend

tool 'tool2'
      ent p0:- ;address of 'aroutine' in p0
      ...
      gos p0,(i3 i5 f3 p2:l1 d4) ;calls 'aroutine' in tool1
      ...
      ret
toolend
```

Example 2

```
;use lookup table
      cpy.p [(i2p (i0*4))+routine_table],p0     ;load address from table
      gos p0,(i3 i5 f3 p2:l1 d4)
      ...
aroutine:
      ent i0 i1 f0 p0:l0 d0
      ...
      ret
aroutine2:
      ent i0 i1 f0 p0:l0 d0
      ...
      ret
      ...

      .data 4

routine_table:
      dc.p aroutine
      dc.p aroutine2
      dc.p aroutine3
      ...
```

2.7.2 qcall

qcall (a macro using the *qcl* instruction) is used to call a different tool.

```
qcall atool,(i0 p0:i0)
```

qcall has an optional third parameter that affects how the called tool is loaded:

• No third parameter	Normal call. Called tool is loaded at the same time as the calling tool.
• VIRTUAL	Called tool is searched for and possibly loaded each time the call is made. Much slower, but good for an infrequently called large tool since the called tool does not take any memory until it is needed, and the kernel can free it again afterwards if memory is low. This technique is often used in programs that have a number of subroutines, only one of which will be executing at a time.
• VIRTUAL+FIXUP	Called tool is searched for and possibly loaded the first time this call is executed, but is then kept in memory. Slow on the first call but nearly as fast as normal calls thereafter.

Example:

```
qcall app/mydirectory/tool2,(p0:-),VIRTUAL
```

2.7.3 ncall

ncall is used to invoke a method on an object. For example,

```
ncall p3,amethod,(p3 p7 i2:i3)
```

invokes the method "amethod" on the object *p3*. By convention, the object pointer is also passed as the first parameter to the method.

The *ncall* macro finds the class tool for the object (in *[p3+ob_class]*) and attempts to invoke the named method in that class. If the class does not have such a method, execution falls through to the class's default method, that typically uses the *parentclass* macro to attempt to invoke the same method in the parent class, and so on until the method is found.

2.7.4 ccall

ccall is used to invoke a method in a particular class on an object.

Example

```
cpy.p [p3+ob_class],p4
ccall p4, amethod,(p3 p7 i2:i3)
```

This example does the same as the *ncall* example above, but this time the the class tool pointer has been explicitly loaded into *p*4.

ccall can be used as an approximate equivalent of a non-virtual invoke, since you specify the class whose amethod method you want to use. However, like in *ncall*, if the method is not found, the default method in the class typically chains to the parent class.

2.7.5 pcall

pcall is used within a class tool to invoke a method in the parent class.

Example

```
method amethod
ent p0 p1 i0:i0
pcall amethod,(p0 p1 i0:i0)   ;invoke parent's amethod method
ret
```

2.7.6 parentclass

parentclass is used in a default method (an *entd* block) to chain to the same method in the parent class. The typical code for a default method in any class that inherits from another class (i.e. it has a parent) is:

```
entd
parentclass
ret
```

2.8 Addresses

VPcode memory is byte addressed. Multi-byte memory accesses load and store data in little endian format (see sections 2.9 and 2.11).

Valid addresses are 32 bit integers. This addressing system permits large addressing ranges and yet permits operation on 32, and 64 bit address range machines with no loss of performance.

Zero and addresses around 0 are considered invalid addresses. Value 0 is NULL pointer. Integers in the range -128 to +127 are not legal addresses in the VPcode system and will never be allocated by a system memory allocation routine. This ensures that accidental access to the region around 0 can be detected. It also provides a number of 'rogue values' that can be safely used for other purposes.

Memory accesses must be naturally aligned, e.g. 32 bit integer accesses must be on 4 byte aligned boundaries. *cpy* and *ld* provide exceptions to this for parsing arbitrary data streams. However, system memory structures are always allocated on appropriate boundaries.

2.9 Number Formats

Data Type Representation

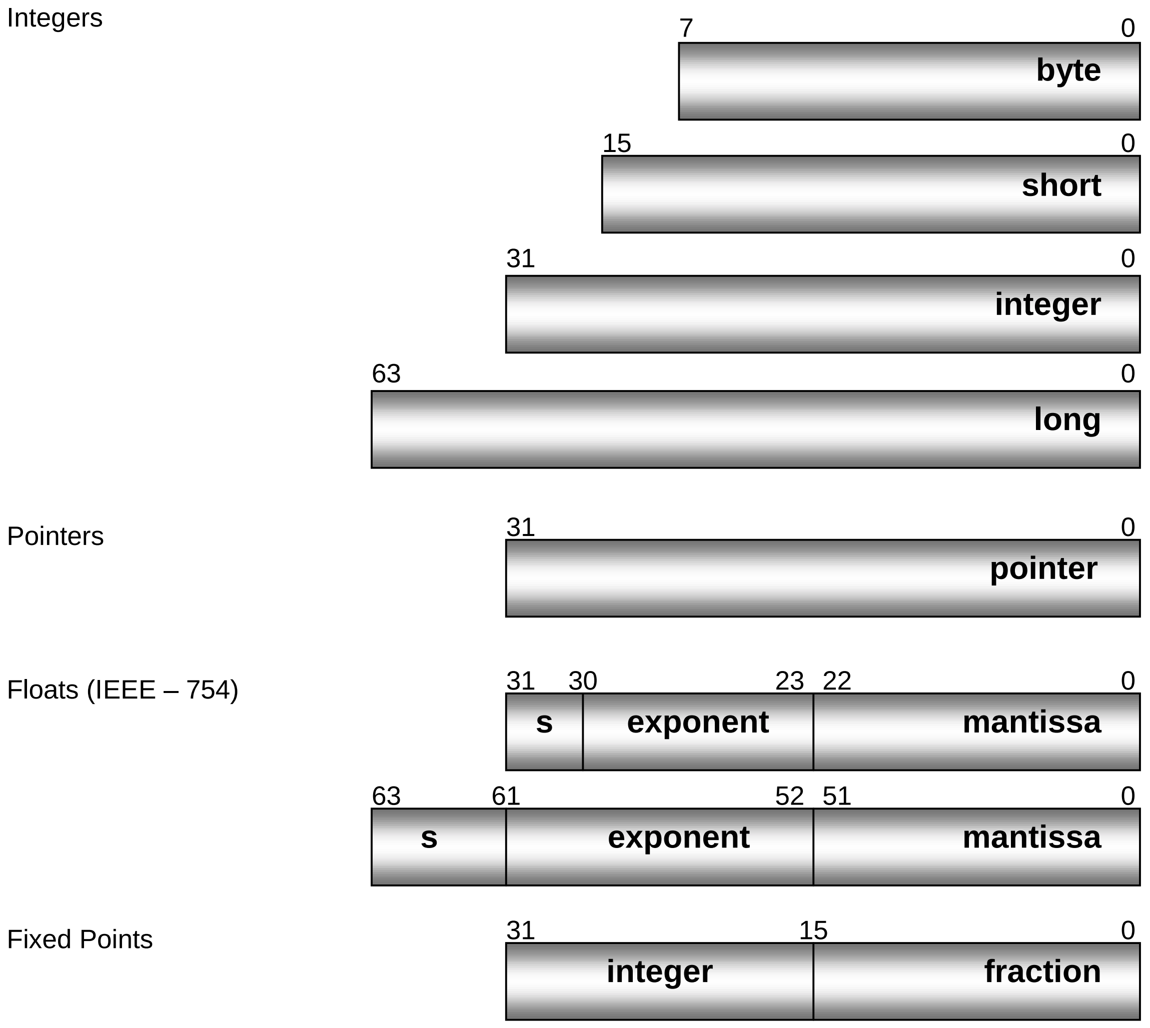

Diagram 2. Data Types.

VPcode integer types are standard 2's complement data types. Integers may also be treated as unsigned. 32-bit integers (referred to as 'integer' or '.i') are the default words of VPcode.

In total, 8 types of number are supported by VPcode:

Integers:	**Type Suffix**
8-bit bytes	.b
16-bit short integers	.s
32-bit integers	.i
64-bit longs	.l

Pointers:	**Type Suffix**
32-bit pointers	.p

Floats:	**Type Suffix**

32-bit floats	.f
64-bit doubles	.d

32-bit 16.16 format is only different from normal 32 bit integer in multiply and divide, when its use is implied by the special expression ops *mulh*, *divh*, and *remh*.

The default type is 32 bit integer. Types can be indicated to the assembler by using the type suffix. There is no need to use the type suffix where the target operand is a register, or the source operand is just a register rather than an expression. Otherwise, it is recommended that a suffix be used.

Example

```
cpy.p 0,p0
cpy p1,p2   ;first field is register defining type
cpy.f [p1],f2
```

Integers are stored as 2's complement binary numbers. They may be considered signed or unsigned unless specifically noted.

Overflow is not detected. The low 32 bits (64 bits for .l) of the result are written to the destination register on an arithmetic add, subtract or multiply.

Memory is considered to be *little endian*, i.e. the low address accesses the least significant byte of the integer, long or float.

Bytes occupy only one byte of memory, but when loaded into a register occupy the whole register, the top 24 bits being zero.

Shorts occupy two bytes of memory. When loaded into a register the top 16 bits are zero.

To load with sign extend, use byte to integer conversion (*b2i*) or short to integer conversion (*s2i*) immediately following a memory load. This will often be optimised away by a peephole optimiser if the processor can do so in a single instruction.

Example

```
cpy (b2i [p0]),i1
```

2.10 Floating Point

VPcode stores floats in both memory and registers in little endian IEEE-754 format. Support for this includes:

- 32 bit (float) and 64 bit (double) accuracy
- support for denormals
- standard meaning for numbers with exponent 0
- standard encodings for not a number

As some CPUs with floating point hardware do not provide support for all of this directly in hardware, one of the *entflags* bits provides a way of switching back to a less accurate but faster mode.

Note that even on a processor with no float support, floating operations will be available via emulation therefore all VPcode programs will be able to run on all systems. They may, however, run slightly slower on systems without suitable hardware.

32 bit float format:

S	Exponent	Mantissa
31	30 23	22

For cases where the exponent is not 0 or $FF, the value represented is:

```
(-1)^S * 2^(E-127) * (1.M)
```

64 bit double format:

S	Exponent	Mantissa
63	62 52	51

For cases where the exponent is not 0 or $7FF, the value represented is:

```
(-1)^S * 2^(E-1023) * (1.M)
```

In these two expressions, S is the sign bit, E the exponent, and M the mantissa. In this expression ^ represents "raised to the power", and 1.M is a number between 1 and 2 with all mantissa bits as fraction bits. See the IEEE standard for more detail.

Special cases of exponent = 0 represents either 0 or tiny value (with appropriate sign). Some VPcode implementations may treat all tiny values as 0, others may maintain them as "denormalised" numbers that are smaller than all other representable values. Implementations are recommended to use whichever is the fastest.

Special case of maximum exponent ($FF or $7FF) is used for infinity with appropriate sign (if mantissa = 0) and not a number (NaN) if mantissa is not equal to 0. (See section on NaN).

2.10.1 Rounding

Some conversion instructions (*d2lt, d2it, f2it*) truncate towards zero; other operations must round to the nearest value.

The case of exactly half way between rounding up and rounding down conforms to IEEE behaviour. This means it is resolved by a tie break of 'round to even': i.e. the result is rounded up if the least significant bit of the result is 1 and down if 0.

```
...R R R R R L G X
----result---| | |____ X = OR of all bits below G
   | |______ G = Guard bit, first non-result bit
   |________ L = least significant bit of result

G=0 X=0: Exact result.
G=0 X=1: Round down.
G=1 X=0: Half way: tie break. Preferably round up if L=1, down if L=0.
G=1 X=1: Round up.
```

Again, setting the appropriate *entflags* bits allows a different method to be used.

2.10.2 NaN

On systems that detect Not-a-Number (NaN) conditions, NaNs will be represented in IEEE-754 format. It should be noted that different hardware will produce different codes for the same NaN condition.

NaNs should be detected using the *ord* and *uno* instructions.

2.11 Fixed Point

The .h ('halfway') fixed point form allows a 16 bit signed integer and 16 bits of fraction (16.16) to be stored in a 32 bit 2's complement value.

It is equivalent to integer * 65536.

add, sub, and *compares* can be done with these signed values using normal integer operations.

divh and *mulh* are provided for division and multiplication. These instructions operate on normal integer registers.

Conversions to and from integers can be done using shifts by 16 bits. Conversions to and from floats can be done by converting using *i2f* and then scaling by 1/65536 or vice versa.

2.12 Little Endian

For consistency, VPcode defines the endianness of the machine - i.e. which order the bytes are stored in the integers and longs in memory. There are numerous ways of encoding - however only two are likely candidates:

- placing the little end (least significant byte) first, or
- placing the big end (most significant byte) first.

VPcode memory is defined to be *little endian*, i.e. the low address accesses the least significant byte of the integer, long or float.

Regardless of underlying hardware, VPcode translators will provide a little endian model.

Note

On big endian processors, a technique called address translation is usually performed by the translator to give the effect of little endian addressing without affecting the order of data in a stored integer. This leaves the natural word size of the machine (usually VPcode integers) ordered the same. (These are frequently accessed and efficiency is paramount.)

The order that VPcode accesses the bytes or shorts in an integer is reversed simply by inverting the low 2 address bits before the access occurs. This operation is transparent to the programmer unless they are writing native code or device drivers that use DMA. Refer to processor specific documentation for details.

2.13 printf and tracef macros

Both macros use a format string followed by a variable number of arguments. There must be exactly the same number of arguments as there are format commands, and the format commands and the arguments are matched in order.

- *printf* writes to stdout.
- *tracef* writes to the TRACE device.

```
 printf "==Hello world==\n" ;no arguments
```

will output

```
==Hello world==
```

```
printf "%d,%d,%d\n",1,2,3 ;with arguments
```

will output 1,2,3

Format commands:

%c	Character
%d,i	Signed integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Float or double
%g	Uses %d,%e or %f, whichever gives full precision in the minimum space
%G	Uses %d,%E or %f, whichever gives full precision in the minimum space
%o	Unsigned Octal
%s	String characters
%u	Unsigned integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays pointer
%n	Pointer to integer that will be updated with number of characters written so far. No argument converted.
%%	Prints a % sign

3. Detailed descriptions

This section consists of an alphabetical listing with detailed descriptions of all VPcode instructions and expression elements. It also includes entries for the VPcode extension and structural macros.

3.1 VPcode Instructions A

3.1.1 abs (macro)

Synopsis

```
abs operand
```

Operation

```
operand <- absolute value of (operand)
```

Description

The absolute value of the operand is placed in the operand.

Valid types are: .i, .l, .f, .d

Macro Expansion

```
if%s %1 < 0
      cpy%s (%1*(-1)),%1
endif
```

Example

```
cpy.f -10.356,f0
abs.f f0
printf "Absolute value = ",f0
```

Result:

```
Absolute value = 10.356000
```

3.1.2 add (expression element)

Synopsis

```
a add b
```

Operation

```
return a + b
```

Description

Adds the two parameters and returns the result. The assembler allows the use of the symbol "+" instead of *add*.

Valid types are: .i, .l, .f, .d, and .p

In the case of 'add pointer', the second parameter is of type integer.

This operation is commutative except in the case of pointers.

Example 1

This example increments the pointer.

```
      cpy.p intcpy,p2
      repeat
            cpy [p2],i1 ;get value
            printf "%d,",i1
            cpy (p2 add 4),p2 ;point to next value
      until.p p2>=intcpyend ;end of table
      printf "End\n",i1
      ...
      ret

      .data 4

intcpy:
      dc.i 0,0,0,-1,-1,1,1

intcpyend:
```

Result:

```
 0,0,0,-1,-1,1,1, End
```

Example 2

The second example adds two integers.

```
cpy 4,i1
cpy (i1 + 3),i3
printf "%d + 3 = %d\n",i1,i3
```

Result

4 + 3 = 7

See Also

add (macro) div div (macro) divh divu mul mul (macro) mulh sub sub (macro)

3.1.3 add (macro)

Synopsis

```
add source,destination
```

Operation

```
destination <- destination + source
```

Description

Adds source and destination and stores the result in destination.

Valid types are: .i, .l, .f, .d, and .p

In the case of 'add pointer,' source is of type integer.

Macro Expansion

```
cpy%s (%2 add %1),%2
```

Example 1

This example increments the pointer.

```
      cpy.p intcpy,p2
      repeat
            cpy [p2],i1 ;get value
            printf "%d,",i1
            add.p 4,p2 ;point to next value
      until.p p2>=intcpyend ;end of table
      printf "End\n",i1
      ...
      ret

      .data 4

intcpy:
      dc.i 0,0,0,-1,-1,1,1

intcpyend:
```

Result:

```
0,0,0,-1,-1,1,1, End
```

Example 2

The second example adds two double values.

```
cpy.d 1.345,d0
cpy.d 2.345,d1
add.d d0,d1 ;result in d1
printf "d0 = %f\td1 = %f\n",d1
```

Result:

```
d0 = 1.345000 d1 = 3.690000
```

See Also

add div div (macro) divh divu mul mul (macro) mulh sub sub (macro)

3.1.4 als (instruction)

Synopsis

```
als constant
```

Operation

```
sp <- sp - framesize(constant)
```

Description

Allocates additional space on the stack for working variables or buffer space within the routine.

Ideally, only small areas should be allocated; any large buffers (over 100 bytes) should be allocated via the *lib/malloc* function call.

1. Within an *ent* block, *als* instructions must be statically matched and nested in *als +N ... als -N* pairs. The exception is that the end of the *ent* block implicitly supplies all the *als -N* instructions needed to complete any open pairs.

2. Any transfer of control that is not a call must not enter or leave any *als* pairs. It can skip over *als* pair(s) entirely.

3. *ret* can appear inside *als* pairs, since the stack is tidied at run time. *ret* does not affect the static analysis of *als* pairs in rule 1.

4. *als* allocates at least the size requested, but the actual size allocated is undefined. This means that, when inside multiple nested *als* pairs, attempting to access the data allocated by an outer *als* using an offset from *sp* gives undefined behaviour. You must store *sp* into a pointer register after the outer *als* instruction and use that pointer register to access the data while execution is within an inner *als* pair.

Example

```
      cpy.p format_string,p0
      als 8 ;allocate stack
      cpy.p sp,p1
      cpy 2,[p1+4] ;put arguments on stack
      cpy 1,[p1+0]
      qcall lib/printf,(p0:i0) ;lib/printf takes arguments from stack
      als -8 ;tidy stack
      ...
      ret

      .data

format_string:
      dc.b "%d %d",LF,0
```

Result:

```
1 2
```

See Also

ent ret allocmem

3.1.5 and (expression element)

Synopsis

```
(a and b)
```

Operation

```
a BITWISE AND b
```

Description

Bitwise *and*. The two parameters are bitwise *and*ed and the result is returned. Matching bits set to 1 remain set to 1, otherwise they are set to 0.

Valid types are: .i and .l. *and* can also be written as &.

This operation is commutative.

Example

```
cpy 100, i1
cpy (i1 and 4),i3
printf "i3=%d\n",i3
```

Result:

```
i3=4
```

See Also

and (macro) and conditional

3.1.6 and (macro)

Synopsis

```
and source,destination
```

Operation

```
destination <- destination BITWISE AND source
```

Description

Bitwise *and*. *source* and *destination* are compared bit by bit. Bits set to 1 in destination remain set to 1 if the matching bit in source is also set, otherwise they are set to 0.

Valid types are: .i, .l

Macro Expansion

```
cpy%s (%2 and %1),%2
```

Example

```
cpy 100,i1
and 4,i1
printf "i1 = %d\n",i1
```

Result:

```
i1 = 4
```

See Also

and, and conditional

3.1.7 and conditional (expression element)

Synopsis

```
a and b
```

Operation

```
a BOOLEAN AND b
```

Description

Condition (boolean) *and*. The two parameters are logically *and*ed and the result is returned. This may also be written as &&.

The assembler encodes this operation as multiple *bc* instructions. This has the effect that if the first parameter is false, the second is not evaluated.

This operation is commutative.

Example

```
      cpy.p str1,p0
      clr i0
      loop
            cpy.b [p0],i1
            bool ((i1 eq NUL) and (i0 ne 0)), exitloop
            inc p0
            inc i0
      endloop

exitloop:
      printf "%d characters read\n",i0
      ...
      ret

 str1:
      dc.b "Example program",0
```

Result:

```
15 characters read
```

An exit from the loop occurs only when both tests are true.

See Also

and and (macro) bcn bcp

3.1.8 asl (macro)

Synopsis

```
asl source,destination
```

Operation

```
destination <- destination SHIFT LEFT BY source
```

Description

Arithmetic shift left. *asl* shifts the bits in destination operand left by the number of places given by the count derived from source. 0 bits are shifted in at the least significant end. Bits shifted out of the most significant end are lost.

Shift values should be in the range 0..31 for integers and 0..63 for long values; other values give undefined results. This is the same as *lsl* (macro).

Valid types are: .i, .l

Macro Expansion

```
cpy%s (%2 lsl %1),%2
```

Example

```
cpy 2,i0
asl 1,i0
printf "i0 = %d\n",i0
```

Result:

```
i0 = 4
```

See Also

asr asr (macro) lsl lsl (macro) lsr lsr (macro)

3.1.9 asr (expression element)

Synopsis

```
(a asr b)
```

Operation

```
a SIGNED SHIFT RIGHT BY b
```

Description

Arithmetic shift right. *asr* shifts the bits in the first parameter right by the number of places given by the second parameter. Sign bits (a copy of the most significant bit) are shifted in from the left hand side. The result is returned.

If the shift count equals or exceeds the data length (32 for integer, 64 for long), or is negative, then the result is undefined. A shift count of 0 is legal.

The second parameter is an integer even for the long version of the instruction.

Valid types are: .i and .l

Example

```
cpy $fff00000,i3 ;keep sign plus exponent
cpy (2 asr i3),i5
printf "i5 = %d\n",i5
```

Result:

```
i5 = 2
```

See Also

asr (macro) asl lsl lsl (macro) lsr lsr (macro)

3.1.10 asr (macro)

Synopsis

```
asr source,destination
```

Operation

```
destination <- destination SIGNED SHIFT RIGHT BY source
```

Description

Arithmetic shift right. *asr* shifts the bits in destination operand right by the number of places given by the count derived from source. Sign bits (a copy of the most significant bit) are shifted in from the left hand side. The result is placed in destination.

If the shift count equals or exceeds the data length (32 for integer, 64 for long), or is negative, then the result is undefined. A shift count of 0 is legal.

The source is an integer even for the long version of the instruction.

Macro Expansion

```
cpy%s (%2 asr %1),%2
```

Example

```
cpy 8,i0
asr 1,i0
printf "i0 = %d\n",i0
```

Result:

```
i0 = 4
```

See Also

asl lsl lsl (macro) lsr lsr (macro)

3.2 VPcode Instructions B

3.2.1 b2i (expression element)

Synopsis

```
(b2i a)
```

Operation

```
return sign_extended_to_integer_from_byte (a)
```

Description

b2i converts a byte to an integer by sign extending.

Note

To zero extend a byte, an *and* may be used. After *cpy.b[p0],i0* the top 24 bits of *i0* are set to zero.

Example 1

```
;sign bit not set
cpy $10001234,i1
cpy (b2i i1),i2
printf "$%x converted to $%x \n",i1,i2
```

Result:

```
$10001234 converted to $34
```

Example 2

```
;sign bit set
cpy $95,i1
cpy (b2i i1),i2
printf "Result = $%x\n",i2
```

Result

```
Result = $ffffff95
```

See Also

cpy.b d_i f_i i_d i_f i_l l_i c2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.2.2 bclr (expression element)

Synopsis

```
(a bclr b)
```

Operation

```
return (a AND NOT (1 << b))
```

Description

Bit clear. A single bit in a copy of the first parameter is cleared. The bit to clear is the bit specified by the second parameter. This value is then returned.

If the bit number is not in the range 0..31 the result is undefined.

Example

```
cpy 186,i1
cpy (i1 bclr 4),i2 ;clear bit 4
printf "%d converted to %d\n",i1,i2
```

Result:

```
186 converted to 170
```

See Also

bcr bset bst

3.2.3 bc (instruction)

Synopsis

```
bcn condition, label
bcp condition, label
```

Operation

```
If condition then PC <- tag
```

Description

Branch conditional. The expression is evaluated and if TRUE a branch occurs to the specified tag.

Unconditional branches may be performed with *go*.

Two variants of *bc* exist:

bcp that implies the branch is probable
bcn that implies the branch is not probable

The choice of *p* or *n* does not alter the meaning of the instruction, but it does permit optimisation by the translator.

Example

```
      clr i1
      cpy 25,i0
startloop:
      inc i1
      dec i0
      bcp (i0 gt 0),startloop
endloop:
      printf "Exit from loop. i1 = %d\n",i1
```

Result:

```
Exit from loop. i1 = 25
```

Using *bcn* will give the same result for the above example.

See Also

go

3.2.4 bcr (macro)

Synopsis

```
bcr source,destination
```

Operation

```
destination <- destination AND (NOT (2^source))
```

Description

bcr clears a single bit in *destination*. *source* holds a value that denotes the bit number (not a mask) to clear. Bit 0 is the least significant bit. The most significant bit of a byte is bit 7 and, of an integer, bit 31.

If the bit number is not in the range 0..31 the result is undefined.

No type suffixes are allowed.

Macro Expansion

```
cpy%s (%2 bclr %1),%2
```

Example

```
cpy 186,i1
cpy i1,i2
bcr 4,i1 ;clear bit 4
printf "%d converted to %d\n",i2,i1
```

Result:

```
186 converted to 170
```

See Also

bclr bset bst

3.2.5 bit (expression element)

Synopsis

```
(a bit b)
```

Operation

```
IF (a AND (2^b)) !=0 THEN true ELSE false
```

Description

Test bit. The bit number specified by the second parameter is examined in the first parameter. If this is set, TRUE is returned; if not set, FALSE is returned.

b can be a constant, register or expression. The result is undefined if *b* is outside the range 0..31.

The returned value can be used immediately in *bcn/bcp* or it can be converted to an integer for further calculation, using *c2i*.

Example

```
      cpy 24,i1
      bcn (i1 bit 4),bitset
      printf "FALSE\n"
      go exit

bitset:
      printf "TRUE\n"

exit:
```

Result:

```
TRUE
```

See Also

c2i bcn bcp nbit

3.2.6 blk (directive)

Synopsis

```
blk count,expression
```

Description

This directive outputs a block constant data into the VP tool, acting as if the *dc* directive had been used <*count*> times, with the expression <*expression*>.

See Also

dc

3.2.7 bool (macro)

Synopsis

```
bool %1,%2
```

Operation

```
If condition then PC <- tag
```

Description

Bool is a primitive macro that takes a boolean expression and makes a jump to a defined label if the result is TRUE.

A size suffix can be used on the *bool* macro.

Valid types are: .i,.l,.d and .f.

To perform a byte comparison, it is possible to use the non-register suffices *.b .s .ns .ni .nl* on *bool.*

For example,

```
bool.b [p0]='A',tag
```

Example

```
      cpy.p str1,p0
      loop
            cpy.b [p0],i1
            bool i1 = NUL, exitloop
            inc p0
      endloop

... exitloop:

      printf "Exit from loop when i1 = NUL\n"

... ret
.data
str1:
dc.b "Example program",0
```

Result:

```
Exit from loop when i1 = NUL
```

See Also

notbool break breakif continue continueif for next if else elseif endif loop endloop repeat until while endwhile

3.2.8 break (macro)

Synopsis

break

Operation

```
PC <- a
```

Description

break causes an unconditional jump to the end of the innermost loop. *break* can be used if additional code needs to be executed between a condition and breaking out of a loop. For example, setting a different flag when there is more than one exit condition. The *breakif* macro is more commonly used.

The following shows the code generated by the *break* macro:

Source code	Generated Code

`repeat`
`loop`
`if...`
	`t0:`
	`t1:`
	`...`

`break`	` go t3`
`endif`	`t2:`
`...`	`...`
`endloop`	`t3:`
`until ...`	`...`

Example

```
      cpy.p str1,p1
      for 20,i0
            cpy.b [p1],i1 ;get character
            if i1=NUL ;end of string
                  break ;exit for loop
            endif inc p1 ;advance to next character
      next i0
      printf "Exit from loop, i1 = %d, i0 = %d\n",i1,i0
      ...
      ret
      .data

str1:
dc.b "Test this once",0
```

Result:

```
Exit from loop, i1 = 0, i0 = 6
```

See Also

breakif bool notbool continue continueif for next if else elseif endif loop endloop repeat until while endwhile

3.2.9 breakif (macro)

Synopsis

breakif condition

Operation

```
If condition then PC <-tag
```

Description

breakif is a conditional break. The break only occurs if the result of the boolean expression is TRUE.

The same rules as those for *bool* apply to this macro.

The following shows the code generated by the *breakif* macro:

Source code	Generated Code
`loop`	`t0:`
`...`	`...`
`breakif`	` bcn (i1 eq $00),t1`
`i1=NUL`	`...`
`...`	`go t0:`
`endloop`	`t1:`

Example

```
      cpy.p str1,p1
      for 20,i0
```

```
            cpy.b [p1],i1 ;get character
            breakif i1=NUL ;break at end of string
            inc p1 ;advance to next character
      next i0
      printf "Exit from loop, i1 = %d, i0 = %d\n",i1,i0
      ...
      ret
      .data

str1:
      dc.b "Test this once",0
```

Result:

```
Exit from loop, i1 = 0, i0 = 6
```

See Also

break bool notbool continue continueif for next if else elseif endif loop endloop repeat until while endwhile

3.2.10 bset (expression element)

Synopsis

```
(a bset b)
```

Operation

```
a OR (1 < b)
```

Description

Bit set. The bit, specified by the second parameter, is set in a copy of the first parameter. This value is then returned.

If the bit number is not in the range 0..31 the result is undefined.

Example

```
cpy 0,i4
cpy (i4 bset 16),i4
printf "i4 = %d\n",i4
```

Result:

```
i4 = 65536
```

See Also

bcr bclr bst

3.2.11 bst (macro)

Synopsis

```
bst source,destination
```

Operation

```
destination <-Destination BITWISE OR (2^source)
```

Description

bst sets a single bit in *destination*. *source* holds a value that denotes the bit number (not a mask) to set.

If the bit number is not in the range 0..31 then the effect is undefined.

No type suffixes are allowed.

Macro Expansion

```
cpy%s (%2 bset %1),%2
```

Example

```
cpy 0,i4
bst 16,i4
printf "i4 = %d\n",i4
```

Result:

```
i4 = 65536
```

See Also

bcr bclr bset

3.3 VPcode Instructions C

3.3.1 c2i (expression element)

Synopsis

```
(c2i a)
```

Operation

```
to_integer_from_conditional(a)
```

Description

Convert conditional (boolean) to integer. Result is 0 for FALSE and 1 for TRUE.

a has to be an expression.

To convert back to condition, use: *ne 0*.

Example

```
cpy (c2i (i1 gt 4)),i2  ;(i1 gt 4) evaluates to TRUE or FALSE
printf "Value of i2=%d\n",i2
```

Result:

```
Value of i2=1
```

if i1 is greater than 4, else i2=0.

See Also

ne b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.3.2 ccall (macro)

Synopsis

```
ccall pointer,name,parameter list
```

Description

ccall attempts to invoke the named method in that class pointed to by the first parameter.

This macro is similar to *ncall* except that the class tool pointer is explicitly loaded into the first parameter.

ccall can be used as an approximate equivalent of a non-virtual invoke, since the class is specified whose method is to be used. However, as for *ncall,* if the method is not found, the default method in the class typically chains to the parent class.

Example

```
cpy.p [p3+ob_class],p4        ;explicitly load the class tool pointer
ccall p4, nameindex, (p3 p7 i2:i3)
```

Result:

Specifies the method (*nameindex*) to use within the class tool pointed to.

See Also

ncall parentclass pcall qcall

3.3.3 ccl (instruction)

ccl is used in the macros *ccall*, *ncall*, *qcall*, *pcall* and *parentclass*; it calls a method in a class, and is not generally used by programmers.

See Also

qcall ccall ncall pcall parentclass

3.3.4 chainclass

Synopsis

```
chainclass classpointer
```

Description

This instruction, which can only be used in an *entd*/*entdr* subroutine, chains onto another class, such as a class's parent class. It is not generally used by programmers.

3.3.5 clr (macro)

Synopsis

```
clr destination
```

Operation

```
destination <-0
```

Description

Clears the destination by setting to zero. Valid types are: .i, .l, .f, .d

Macro Expansion

```
cpy 0,destination
```

Example

```
cpy 2,i1
printf "i1 set to %d\n",i1
clr i1
printf "i1 cleared to %d\n",i1
```

Result:

```
i1 set to 2
i1 cleared to 0
```

See Also

cpy

3.3.6 continue (macro)

Synopsis

```
continue
```

Operation

```
PC <-a
```

Description

continue causes an unconditional jump to the beginning of the innermost loop. *continue* can be used if additional code needs to executed between a condition and restarting a loop cycle. The *continueif* macro is more commonly used.

The following shows the code generated by the *continue* macro:

Source code	Generated Code
loop ... if ... ... continue endif ... endloop	t0: ... bcp (...),t1 ... go t0 t1: ... go t0 t2:

Example

```
      cpy.p str1,p1
      cpy 100,i0
      qcall lib/malloc,(i0:p0)
      if p0 != NULL
            cpy.p p0,p2       ;save pointer to start of buffer
            loop
            cpy.b [p1],i1 ;get character
            if i1='/'   ;ignore this character
                        inc p1
                        continue
            endif
            cpy.b i1,[p0]     ;store character
            breakif i1=NUL
                  inc p1 ;advance
            inc p0
            endloop
            printf "String stored = >%s<\n",p2
      endif
      qcall lib/free,(p2:-)
      ...
      ret
      .data

str1:
      dc.b "testex/break.asm",0
```

Result:

```
String stored = >testexbreak.asm<
```

See Also

continueif bool notbool break breakif for next if else elseif endif loop endloop repeat until while endwhile

3.3.7 continueif (macro)

Synopsis

```
continueif condition
```

Operation

```
If condition then PC <-tag
```

Description

continueif is a conditional continue. The jump is only made if the result of the boolean expression is TRUE.

The same rules as those for *bool* apply to this macro.

The following shows the code generated by the *continueif* macro:

Source code	Generated Code
loop ... continueif i1 = '/' ... endloop	t0: ... bcp (i1 eq $2F),t0 ... go t0 t1:

Example

```
      cpy.p str1,p1
      cpy 100,i0
      qcall lib/malloc,(i0:p0)
      if p0 != NULL
            cpy.p p0,p2 ;save pointer to start of buffer
            loop
                  cpy.b [p1],i1     ;get character
                  inc p1
                  continueif i1='/'       ;ignore this character
                  cpy.b i1,[p0]     ;store character
                  breakif i1=NUL
                  inc p0
            endloop
            printf "String stored = >%s<\n",p2
      endif
      qcall lib/free,(p2:-)
      ...
      ret
      .data

str1:
      dc.b "testex/break.asm",0
```

Result:

```
String stored = >testexbreak.asm<
```

See Also

bool notbool break breakif continue for next if else elseif endif loop endloop repeat until while endwhile

3.3.8 cpbb (instruction)

Synopsis

```
cpbb source,destination,length
```

Operation

```
[destination] <-[source],(block of length bytes)
```

Description

Copy block (byte). A block of memory pointed to by source, with size specified in bytes by length, is copied to location specified by destination.

Undefined values are written if source and destination overlap.

The registers are unchanged after this operation.

Example

```
      cpy.p string2,p0 ;destination
      cpy.p string1,p1 ;source
      cpy 1,i0 ;length
      cpbb p1,p0,i0
      printf "Destination string after copy = >%s<\n",p0
      ...
      ret

      .data

string1:
      dc.b "x",0

string2:
      dc.b "abcdef",0
```

Result:

```
Destination string after copy = >xbcdef<
```

See Also

```
cpbi cpsb
```

3.3.9 cpbi (instruction)

Synopsis

```
cpbi source,destination,length
```

Operation

```
[destination] <-[source],(block of length bytes)
```

Description

Copy block (integer). A block of memory pointed to by *source*, with size specified in bytes by length, is copied to the location specified by *destination*.

All pointers and sizes must be 4-byte aligned.

Undefined values are written if source and destination overlap.

The registers are unchanged after this operation.

Example

```
      cpy 100,i0
      qcall lib/malloc,(i0:p0)      ;new block
      cpy.p p0,p3       ;save pointer
      cpy.p intcpy,p2   ;original block
      cpy.p p2,p4       ;save pointer
      clr i2
      repeat
            cpy [p2],i1       ;get value
            add.p 4,p2  ;point to next value
            inc i2
      until.p p2>=intcpyend   ;end of table
      cpbi p4,p3,i0     ;copy block
      printf "New block now contains: "
      for i2
            cpy [p3],i1       ;get value
            add.p 4,p3  ;point to next value
            printf "%d ",i1
      next i2
      printf "\n"
      qcall lib/free,(p0:-)
      ...
      ret
      .data 4

intcpy:
      dc.i 0,0,0,-1,-1,1,1

intcpyend:
```

Result

```
New block now contains: 0 0 0 -1 -1 1 1
```

See Also

cpbb cpsb

3.3.10 cpsb (instruction)

Synopsis

```
cpsb source, destination, length
```

Operation

```
IF source register = destination register
l <-0
WHILE [s] NOT EQUAL 0 (byte)

      s <-s + 1
      l <-l + 1

END
ELSE

      l <-0

WHILE [s] NOT EQUAL 0 (byte)

      [d] <-[s] (byte)
      s <-s + 1
      d <-d + 1
      l <-l + 1

END [d] <-0 (byte)
```

Description

Copies string (byte). A null terminated byte string pointed to by *source* is copied to the location pointed to by *destination*.

source and *destination* registers are updated to point to the NUL at the end of the string. If a third register is specified, the size of the string is placed in it.

It is explicitly permitted for *source* and *destination* registers to be identical, in which case the length can be returned without a copy taking place.

If the source and destination memory areas overlap undefined behaviour results.

Example

```
      cpy.p strlen,p0   ;string to copy
      cpy.p p0,p1
      cpsb p1,p1,i0     ;advances pointer and returns length
      printf "Length = %d\n",i0
      cpy (i0+1),i0     ;allow for terminator
      qcall lib/malloc,(i0:p2)      ;new location must be large enough
      cpy.p p2,p1       ;save pointer to start of new location
      cpsb p0,p2  ;advances both pointers
      printf "String in new location = '%s'\n",p1
      ...
      ret
      .data

strlen:
      dc.b "76#67$08/",0
```

Result

```
Length = 9
String in new location = '76#67$08/'
```

See Also

cpbb cpbi

3.3.11 cpy (instruction)

Synopsis

```
cpy source,destination
```

Operation

```
destination <-source
```

Description

Copies the value of *source* to *destination*. *destination* can be either a register or memory location. *source* can be a register, a constant, an expression or memory.

Valid types for both register and memory copying are: .d, .f, .i, .l, .p. .i is the default data type.

Types valid only when copying to/from memory are: .b, .s, .ns, .ni, .nl, .nd

Most data types must be naturally aligned:

- .s on a 2-byte boundary
- .f, .i and .p on a 4-byte boundary
- .l and .d on an 8-byte boundary.

The four non-aligned modes provided do not need to be naturally aligned:

- .ns and .ni may be aligned on any byte boundary,
- .nl and .nd may be aligned on any 4-byte boundary

The following table shows the possible combinations of source and destination:

Source	**Destination**		
	Register	**Expression**	**Memory**
Constant	cpy 5,i0 cpy $7f,i1	Not unless memory	cpy 5,[p0] cpy $7f,[p0+4] cpy 5,[p0+i0] cpy 5,[p0+i4+16] cpy 5,[(p0+(i4*4))] cpy 5,[table] cpy 5,[tag expression]
Register	cpy i0,i1	Not unless memory	cpy i0,[p0]

	cpy.d d0,d1		cpy i0,[p0+4] cpy i0,[p0+i0] cpy i0,[p0+i4+16] cpy i0,[(p0+(i4*4))] cpy i0,[table] cpy i0,[tag expression]
Expression	cpy (i0+i1),i0	Not unless memory	cpy (i0+i1),[p0] cpy (i0+i1),[p0+4] cpy (i0+i1),[p0+4] cpy (i0+i1),[p0+i4+16] cpy (i0+i1),[(p0+(i4*4))] cpy (i0+i1),[table] cpy (i0+i1),[tag expression]
Memory	cpy [p0],i0	Not unless memory	cpy [p0],[p1] cpy [p0],[p1+4] cpy [p0],[p1+i0] cpy [p0],[p1+i4+16] cpy [p0],[(p1+(i4*4))] cpy [p0],[table] cpy [p0],[tag expression]

Example

```
      cpy.p str1,p1
      cpy 100,i0  ;constant to register
      qcall lib/malloc,(i0:p0)
      if p0 != NULL
            cpy.p p0,p2       ;register to register
            loop
                  cpy.b [p1],i1     ;memory to register
                  inc p1
                  continueif i1='/'
                  cpy.b i1,[p0]     ;register to memory
                  breakif i1=NUL
                  inc p0
            endloop
            printf "String stored = >%s<\n",p2
      endif
      qcall lib/free,(p2:-)
      ...
      ret
      .data

str1:
      dc.b "testex/break.asm",0
```

Result:

```
String stored = >testexbreak.asm<
```

See Also

ld st

3.4 VPcode Instructions D

3.4.1 d_i (instruction)

Synopsis

```
d_i source,destination
```

Operation

```
destination <-BITWISE source
```

Description

Moves bit pattern from a double register to a pair of integer registers. *source* specifies a double register. The bit pattern is moved into the integer register specified by *destination* and the next sequential register (n and n+1).

Example

```
      cpy.d 4.0,d0

d_i d0,i0   ;bit pattern moved into i0 and i1
      printf "$%08x, $%08x\n",i1,i0
```

Result:

```
$40100000, $00000000
```

See Also

f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.4.2 d2f (expression element)

Synopsis

```
d2f a
```

Operation

```
double to float (a)
```

Description

Converts 64 bit double value to 32 bit float.

Example

```
cpy.d 123.8956,d0
cpy (d2f d0),f0
printf "%f\n",f0
```

Result:

```
123.895599
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.4.3 d2ir (expression element)

Synopsis

```
d2ir a
```

Operation

```
round to integer (a)
```

Description

Converts 64 bit double value to integer, rounding to the nearest value.

Example

```
cpy.d 123.8956,d0
cpy (d2ir d0),i0
printf "%d\n",i0
```

Result:

```
124
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.4.4 d2it (expression element)

Synopsis

```
(d2it a)
```

Operation

```
truncate to integer (a)
```

Description

Converts 64 bit double value to integer, truncating towards 0.

Example

```
cpy.d 123.8956,d0
cpy (d2ir d0),i0
printf "%d\n",i0
```

Result:

```
123
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.4.5 d2lr (expression element)

Synopsis

```
d2lr a
```

Operation

```
round to long (a)
```

Description

Converts 64 bit double value to 64 bit long, rounding to nearest.

Example

```
cpy.d 123.8956,d0
cpy (d2lr d0),l0
printf "%ld\n",l0
```

Result:

```
124
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.4.6 d2lt (expression element)

Synopsis

```
d2lt a
```

Operation

```
truncate to long (a)
```

Description

Converts 64 bit double to 64 bit long, truncating towards 0.

Example

```
cpy.d 123.8956,d0
cpy (d2lr d0),l0
printf "%ld\n",l0
```

Result:

```
123
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.4.7 dat (instruction)

Data section starts. This instruction is not usually used by the programmer; the assembler macro is .data.

See Also

.data

3.4.8 .data (macro)

Synopsis

```
.data n
```

Operation

```
define_start_of_data_section
```

Description

.data introduces some *n* aligned data, where *n* is optional and defaults to 1. This data can be placed in the middle of a tool, with the .code macro used to switch back to executable code.

All data introduced by the *.data* directive will be put at the end of the tool in the order 8 aligned, 4 aligned, 2 aligned, unaligned to minimise wastage by alignment.

Example

```
      cpy.p data_string,p0
      printf "%s\n",p0
      ...
      ret

      .data

._string:
      dc.b "hello",0
```

Result:

```
hello
```

See Also

sync syncreg

3.4.9 dc (directive)

Synopsis

```
dc expression
```

Operation

```
define_literal_block_ of_data
```

Description

dc defines a literal block of data. This should usually only be used in the data section. The terminating NUL is required to terminate strings.

Example

```
      cpy.p hello,p0
      printf "%s\n",p0
      ...
      ret

      .data

hello:
      dc "Hello world",0
```

Result:

```
Hello world
```

See Also

.data dc.i dc.d dc.p

3.4.10 dc.d

Synopsis

```
dc.d
```

Operation

Define a literal double.

Description

dc.d defines one or more double values in the data section.

Example

```
      cpy.p dc_table,p0
      cpy.p dc_end,p1
      while.p p0 < p1
            cpy.d [p0],d0
            printf "Value = %f\n",d0
            cpy.p (p0+8),p0
      endwhile
      ...
      ret
      .data 8

dc_table:

      dc.d 1.4
      dc.d -3.0
      dc.d 8.234

dc_end:
```

Result:

```
Value = 1.400000
Value = -3.000000
Value = 8.234000
```

See Also

.data dc dc.i dc.p

3.4.11 dc.i

Synopsis

```
dc.i
```

Operation

Define a literal integer.

Description

dc.i defines one or more integer values in the .data section.

Example

```
      cpy.p dc_table,p0
      cpy.p dc_end,p1
      while.p p0 < p1
            cpy [p0],i0
            printf "Value = %d\n",i0
            cpy.p (p0+4),p0
      endwhile
      ...
      ret
      .data 4

dc_table:
      dc.i 4
      dc.i 0
      dc.i 8

dc_end:
```

Result:

```
Value = 4 Value = 0 Value = 8
```

See Also

.data dc dc.d dc.p

3.4.12 dc.p

Synopsis

```
dc.p
```

Operation

Define pointer to block of data.

Description

dc.p is used when it is necessary to specify a tag in a block of data that can be a code tag or a data tag.

Example

```
      stdout p0
      cpy.p fmt_string,p1
      cpy.p arg_list,p2
      qcall lib/vfprintf,(p0 p1 p2:i0)
      printf "%d chars were printed\n",i0
      ...
      ret
      .data

fmt_string:
      dc.b">%f< >%s< >%d<",LF,0
      .align 8

arg_list:
      dc.d 1234.567           ;8 bytes for double
      dc.p string_param2      ;pointer to string
      dc.i 4321         ;4 bytes for integer
      string_param2:
            dc.b "hello world",0
```

Result:

```
>1234.567000< >hello world< >4321<
37 characters were printed
```

See Also

.data dc dc.d dc.i

3.4.13 dec (macro)

Synopsis

```
dec operand
```

Operation

```
operand <-operand - 1
```

Description

One is subtracted from the operand and the result is placed in the operand.

Valid types are: .i, .l, .f, .d, .p

Macro Expansion

```
cpy%s (%1 - 1),%1
```

Example

```
cpy 10,i0
dec i0
print "i0=%d\n",i0
```

Result:

```
i0=9
```

See Also

inc sub

3.4.14 defbegin (directive)

Synopsis

```
defbegin (0)
```

Operation

None

Description

Start a block. This macro starts a new name scope for VPcode registers. Definition blocks can be nested. The top level (or outermost) block is one that is not enclosed by any other block. Using the 0 parameter forces the block to the top level.

Names can be reused in different blocks but cannot be redefined in inner blocks.

Example

```
;this macro can only be used in ent blocks
;using the def macros to keep track of the registers.
.macro shwz

      .check %n=2; shwz value,count
      cpy 0,%2
      if (%1 ne 0)
            defbegin
            ;use names in macros that won't be used outside
            defi shwz_shft,shwz_mask
            cpy $10,shwz_shft
            cpy -1,shwz_mask
            repeat
                  lsl shwz_shft,shwz_mask
                  if ((%1 and shwz_mask) eq 0)
                        lsl shwz_shft,%1
                        add shwz_shft,%2
                  endif
                  lsr 1,shwz_shft
            until shwz_shft = 0

            ;the defend automatically zaps shwz_shft,_mask
            ;and frees the registers for further use

            defend

      endif

.endm

tool 'defdemo',VP, F, MAIN,1024,64

;Assembler will calculate the work registers used

      ent -:-

;the defbegin 0 checks that it is at the outer level
;and start a block with named registers.

      defbegin 0
      defp argv   ; this will be p0
      defi argc,val1,val2,cnt       ; these are i0,i1,i2,i3
      qcall lib/argcargv, (-:argv argc )
      cpy $01234567,val1
      cpy val1,val2
      shwz val2,cnt
      printf "original %X shwz %X count %d\n",val1,val2,cnt

; the returns can be left as VP registers if different but they could have
a defset as follows

      defset retcode,argc
      cpy 0,i0
      qcall lib/exit, (i0:-)
      ret

 ; The outermost defend doesn't do a zap
 ; The 0 is to check it really is matched

      defend 0

toolend

.end
```

See Also

defend defendnz defzap defi defl deff defd defp defset

3.4.15 defd (directive)

Synopsis

```
defd name
```

Operation

None

Description

Define names for double registers.

Example

```
defbegin 0
defd abc,def ; abc is d0 and def is d1
```

See Also

defbegin defend defendnz defzap defi defl deff defp defset

3.4.16 defend (directive)

Synopsis

```
defend (0)
```

Operation

```
zap register
```

Description

End a block. End a name scope for VPcode registers. The names become undefined. The parameter 0 can be specified to check this operation ends at the top level - that there are no missing *defend*s.

The registers are zapped except if the outer level is reached.

Note

If a *go* at the end of a block is to an outer *defbegin-defend* block then *defend* can be put before the *go* rather than after it thus helping code optimisation.

Example

See defbegin for example.

See Also

defbegin defendnz defzap defi defl deff defd defp defset

3.4.17 defendnz (directive)

Synopsis

```
defendnz
```

Operation

None

Description

End a block without any zaps. End a name scope for VPcode registers.

defendnz differs from *defend* in that it never zaps the registers and can therefore be used after a *go* or *ret* without putting in spurious zaps.

Example

See defend for *example.*

See Also

defbegin defend defzap defi defl deff defd defp defset

3.4.18 deff (directive)

Synopsis

```
deff name
```

Operation

None

Description

Define names for float registers.

Example

```
defbegin 0
deff abc,def ; abc is f0 and def is f1
```

See Also

defbegin defend defendnz defzap defi defl defd defp defset

3.4.19 defi (directive)

Synopsis

```
defi name
```

Operation

None

Description

Define names for integer registers.

Example

See *defbegin* for example.

See Also

defbegin defend defendnz defzap defl deff defd defp defset

3.4.20 defl (directive)

Synopsis

```
defl name
```

Operation

None

Description

Define names for long registers.

Example

```
defbegin 0
defl abc, def; abc is l0 and def is l1
```

See Also

defbegin defend defendnz defzap defi deff defd defp defset

3.4.21 defp (directive)

Synopsis

```
defp name
```

Operation

Define names for pointer registers.

Description

Define names for pointer registers.

Example

```
defbegin 0
defp abc, def; abc is p0 and def is p1
```

See Also

defbegin defend defendnz defzap defi defl deff defd defp

3.4.22 defset (macro)

Synopsis

```
 defset a,b
```

Operation

Equate a name to a register or number.

Description

Block structure equate a name to a register or number.

```
defbegin
defi abc
defset xxx,abc
defset yyy,i3
...
defend
```

xxx is set to the same integer register as *abc*. *yyy* is set to i3.

xxx and *yyy* are not zapped at the *defend* but *abc* will be. They will however be set undefined.

Example

See *defbegin* for example.

See Also

defbegin defend defendnz defzap defi defl deff defd defp

3.4.23 defzap (macro)

Synopsis

```
defzap
```

Operation

Zap the registers for the current block

Description

This macro zaps the register of the current block before a *go* to an outer block. The block is not ended, the registers stay defined.

Example

See *defbegin* for example.

See Also

defbegin defend defendnz defi defl deff defd defp defset

3.4.24 div (expression element)

Synopsis

```
(a div b)
```

Operation

```
return a/b (signed)
```

Description

Signed division. The first parameter is divided by the second parameter using signed arithmetic and the result is returned.

For integer and long types, dividing by zero causes the processor exception EXC_INT_DIV_0, which if uncaught becomes a SIGFPE signal. Dividing the minimum integer/long (i.e. a 1 in the top bit and 0 in all other bits) by -1 gives the minimum integer/long as its result. This is not the case for dividing a float or a double by zero.

The assembler allows the use of the symbol "/" instead of *div*.

Example 1

```
;integer divide
cpy 7,i0
cpy 3,i1
cpy (i0 div i1),i2
printf "%d div %d = %d\n",i0,i1,i2
```

Result:

```
7 div 3 = 2
```

Example 2

```
;double float divide
cpy.d 7.0,d0
cpy.d 3.0,d1
cpy.d (d0 div d1),d2
printf "%f div %f = %f\n",d0,d1,d2
```

Result

```
7.000000 div 3.000000 = 2.333333
```

See Also

add add (macro) div (macro) divh divu mul mul (macro) mulh sub sub (macro)

3.4.25 div (macro)

Synopsis

```
div source,destination
```

Operation

```
destination <-destination/source(signed)
```

Description

Signed division. *source* is divided by *destination* using signed arithmetic and the result is stored in *destination*.

For integer and long types, dividing by zero causes the processor exception EXC_INT_DIV_0, which if uncaught becomes a SIGFPE signal. Dividing the minimum integer/long (i.e. a 1 in the top bit and 0 in all other bits) by -1 gives the minimum integer/long as its result. This is not the case for dividing a float or a double by zero.

Macro Expansion

```
.if '.%s'='..H' | '.%s'='..h'
      cpy.i (%2 divh %1),%2
.else
      cpy%s (%2 div %1), %2
.endif
```

Example 1

```
cpy 7,i0
cpy 3,i1
div i1,i0
printf "i0=%d\n",i0
```

Result:

```
i0=2
```

Example 2

```
cpy.d 7.0,d0
cpy.d 3.0,d1
div d1,d0
printf "Result = %f\n",d0
```

Result:

```
Result = 2.333333
```

See Also

add add (macro) div divh divu mul mul (macro) mulh sub sub (macro)

3.4.26 divh (expression element)

Synopsis

```
(a divh b)
```

Operation

```
(a * 65536)/b (signed)
```

Description

Fixed point division. Divides the first parameter by the second using fixed point arithmetic. The effect is to scale the first parameter by shifting it left by 16 bits (into a 48 bit word) before doing an integer divide.

Valid types are: .i

Overflow or dividing by zero yields undefined behaviour.

Example

```
cpy 7,i0
cpy 3,i1
cpy (i0 divh i1),i2
printf "%d divh %d=%d\n",i0,i1,i2
```

Result:

```
7 divh 3 = 152917
```

See Also

add add (macro) div div (macro) divu mul mul (macro) mulh sub sub (macro)

3.4.27 divu (expression element)

Synopsis

```
(a divu b)
```

Operation

```
return a/b (unsigned)
```

Description

Unsigned division. The first parameter is divided by the second parameter using unsigned arithmetic and the result returned.

Valid types are: .i and .l

Dividing by zero causes a SIGFPE signal.

Example

```
cpy 7,i0
cpy 3,i1
cpy (i0 divu i1),i2
printf "%d divu %d=%hd\n",i0,i1,i2
```

Result:

```
7 divu 3 = 2
```

See Also

add add (macro) div div (macro) divh mul mul (macro) mulh sub sub (macro)

3.5 VPcode Instructions E

3.5.1 else (macro)

Synopsis

```
else
```

Operation

```
PC <-a
```

Description

else must be used in conjunction with an *if/endif* structure. It defines an alternative path of code to be executed if none of the previous *if* conditions have been met.

The following shows the code generated by the *else* macro:

Source code	Generated Code
`if i0=0`	`     bcp (i0 ne $00),t0`
`     ...`	`     ...`
`else`	`     go t1`
`     ...`	`t0:`
`endif`	`     ...`
	`t1:`

Example

```
cpy 'A',i0
qcall lib/isdigit,(i0:i1) ;function to test if digit
if i1 != 0
      printf "'%c' is a digit\n",i0
else
      printf "'%c' is not a digit\n",i0
endif
```

Result:

```
'A' is not a digit
```

See Also

if elseif endif bool notbool break breakif continue continueif for next loop endloop repeat until while endwhile

3.5.2 elseif (macro)

Synopsis

```
elseif condition
```

Operation

```
if FALSE then PC <-tag
```

Description

elseif must be used in conjunction with an *if/endif* structure. It defines an alternative block of code, to be executed if the result of its boolean expression is TRUE. It can be used to produce case statements.

The same rules as those for *bool* apply to this macro.

The following shows the code generated by the *elseif* macro:

Source code	Generated Code
`if i0=0`	`     bcp (i0 ne $00),t0`
`     ...`	`     ...`
`elseif i0=1`	`     go t2`
`     ...`	`t0:`
`endif`	`     bcp (i0 ne $01),t1`
	`t1:`
	`t2:`

Example

```
cpy.p string1,p0
cpy.p string2,p1
cpy 6,i0
qcall lib/memcmp,(p0 p1 i0:i0)
if i0<0
      printf "string1 is less than string2\n"
elseif i0=0
      printf "string1 and string2 are equal\n"
else
      printf "string1 is greater than string2\n"
endif
```

```
...
ret

.data

string1:
      dc.b "abcxef",0

string2:
      dc.b "abcdef",0
```

Result:

```
string1 is greater than string2
```

See Also

if else endif bool notbool break breakif continue continueif for next loop endloop repeat until while endwhile

3.5.3 endif (macro)

Synopsis

```
endif
```

Operation

```
end condition
```

Description

endif must be used in conjunction with the *if* macro. It marks the end of the *if* structure.

Each *endif* produces a unique label, and although labels take up space in the VPcode, they do not in the translated code.

The following shows the code generated by the *endif* macro:

Source code	Generated Code

`if i0=0`
`     ...`
`endif`
	`     bcp (i0 ne $00),t0`
	`     ...`
	`t0:`

endif generates the label required by the corresponding *if* and places it into the code.

Example

```
cpy '9',i0
qcall lib/isdigit,(i0:i1)
if i1 != 0
      printf "'%c' is a digit\n",i0
endif
```

Result:

```
'9' is a digit
```

See Also

if else elseif bool notbool break breakif continue continueif for next loop endloop repeat until while endwhile

3.5.4 endloop (macro)

Synopsis

```
endloop
```

Operation

```
PC <-tag generated by corresponding loop
```

Description

endloop must be used in conjunction with the *loop* macro. It marks the end of a *loop* structure.

The following shows the code generated by the *endloop* macro:

Source code	Generated Code
`loop`	`t0:`
`...`	`     ...`
`breakif i1 > 'A'`	`     bcn (i1 eq $41),t1`
`...`	`     ...go t0:`
`endloop`	`t1:`

The second label is generated only if required i.e. something in the loop needs to jump to it.

Example

```
      cpy.p str1,p1
      cpy 100,i0
      qcall lib/malloc,(i0:p0)
      if p0 != NULL
            cpy.p p0,p2       ;save pointer to start of buffer
            loop
                  cpy.b [p1],i1     ;get character
                  inc p1
                  continueif i1='/'       ;ignore this character
                  cpy.b i1,[p0]     ;store character
                  breakif i1=NUL
                  inc p0

            endloop
            printf "String stored = >%s<\n",p2
      endif
      qcall lib/free,(p2:-)
      ...
      ret

      .data

str1:
      dc.b "testex/break.asm",0
```

Result:

```
String stored = >testexbreak.asm<
```

See Also

loop bool notbool break breakif continue continueif for next if else elseif endif repeat until while endwhile

3.5.5 endwhile (macro)

Synopsis

```
endwhile
```

Operation

```
PC <-tag generated by corresponding loop
```

Description

endwhile must be used in conjunction with the *while* macro. It marks the end of a *while* structure.

The following shows the code generated by the *endwhile* macro:

Source code	Generated Code

`while i1!=NUL`
`     ...`
`endwhile`
	`t0:`
	`     bcn (i1 eq $00),t1`
	`     ...`
	`     go t0`
	`t1:`

The second tag is generated only if required i.e. something in the loop needs to jump to it.

Example

```
      cpy.p intcpy,p0 ;start of table
      cpy.p intcpyend,p1 ;end of table
      while p0 != p1 ;exit when p0=p1
            cpy [p0],i1 ;get value
            add.p 4,p0
            printf "%d ",i1
      endwhile ;exit when p0=p1
      printf "\n"
      ...
      ret

      .data 4

intcpy:
      dc.i 0,0,0,-1,-1,1,1

intcpyend:
```

Result:

```
0 0 0 -1 -1 1 1
```

See Also

while bool notbool break breakif continue continueif for next if else elseif endif loop endloop repeat until

3.5.6 ent (instruction)

Synopsis

```
ent P:P
```

Operation

```
create stack frame
```

Description

Entry to routine. This instruction should be specified immediately after a tag or tool start if an entry point is defined by the tool, to indicate a routine start.

It takes:

- a list of input registers
- a list of return registers

Each list must contain register ranges starting at 0 and be contiguous. The assembler permits a range to be entered with register-register and "-" to be entered if there are no registers in that list.

```
ent p0 i0-i3 p1:-
```

The separator between the input and output lists is a colon ':'.

This instruction may allocate a stack frame appropriate to the native processor. Any subsequent *ret* instructions will tidy up the stack frame allocated by the preceding *ent.*

The *ent* block has to be self-contained i.e. it cannot branch outside itself.

After entry

- *sp* points to the "top" of the stack frame, i.e. lowest address
- *pp* points to any parameters passed in on the stack, (invalid if *schk* instruction)
- *lp* contains the return address.

The stack frame allocated is unpredictable and depends upon the platform.

The binary encoding for this instruction also includes a specification of what local registers are used in the subroutine. This is calculated by the assembler, so should not be included in VP source. This is true for all *entx* instructions.

Example

```
      tool 'toolname',VP,TF_MAIN,8092,0 ;tool start
      ent p0 i0-i2: -   ;start of ent block
      ...
      gos routine2,(p3:i0)
      ret

routine2:
      ent p0:i0   ;start of second ent block
      ...
      ret
```

Result:

Input registers are p0, i0 to i2; there are no output registers.

See Also

entd enth entl ret

3.5.7 entd (instruction)

Synopsis

```
entd
```

Operation

```
create stack frame
```

Description

Entry instruction for a default method. The caller can pass any or no parameters. The parameters are not specified in the entd instruction since, for a default method, the parameters are not known at assemble/translate time.

Like a normal *ent* block, the *entd* block gets its own local set of VP registers. No input parameters appear in these local registers, except that the first pointer parameter appears in p0. If *parentclass* is used to pass execution on to another class, p0 is passed on as the first pointer parameter, therefore p0 should not be modified in the *entd* block. This is true even if it is known that there are no pointer parameters in the original call.

Unlike a normal *ent* block, in an *entd* block si is passed in and out as well as p0, and may be used as a normal integer register. If the *entd* was invoked by a method call (*ncall*/*pcall*/*ccall*), then *si* contains the atom of the method name.

Example

```
entd
parentclass
ret
```

Result:

Shows *entd* used in the default method.

See Also

ent entih entl ret

3.5.8 ente (instruction)

Synopsis

```
ente P:P
```

Description

This instruction is similar to *ent,* except that, in addition to the specified formal parameters, a statics pointer is passed in (if called by *gose* or *qcle*). The binary encoding includes a field specifying which pointer register the statics pointer is to be placed in; this is calculated by the assembler, so should not be included in VP source. Through the subroutine, the pointer register containing the statics pointer is accessed as *__ep*. Note that, because the VP register which will be assigned to __ep is not known in pass 1, *?reg(__ep)* returns 0 and *__ep* cannot be used as the argument of *__regnum*.

The formal parameters are the same as in the *ent* directive.

3.5.9 entflags (instruction)

Synopsis

```
entflags <constant>
```

Description

This instruction sets flags governing certain aspects of behaviour for the whole subroutine, and must appear at the start of the subroutine either straight after the *ent*, *entl*, *ente*, *entle*, *entih*, *entd* or *entdr* instruction, or straight after any *schk* instruction which appears after the *ent** instruction.

The C operand is a constant with the following bit settings:

- bit 0: floating point calculation mode; 0 for IEEE-754 mode, 1 for native mode. In native mode, floats and doubles are still stored in memory in IEEE-754 format, but arithmetic operations are not guaranteed to be performed to IEEE-754 accuracy. On some CPUs this increases floating point performance.
- bit 1: non local goto target never returns.
- bit 2: subroutine is not to appear in a debugger stack trace. If the debug information format output by a particular translator contains a flag which may be set to indicate that the subroutine should not appear in a debugger stack trace, then setting bit 2 of *entflags* causes that flag to be set. The setting of bit 2 does not alter the semantics of the VPcode in any way.
- bits 31..3: reserved, should be 0. Setting any of these bits yields an invalid tool.

If no *entflags* instruction appears in a subroutine, the effect is the same as entflags 0.

3.5.10 entih (instruction)

Synopsis

```
entih P:P
```

Operation

```
create stack frame
```

Description

entih is the entry instruction for interrupt handling procedures. It is not used by the application programmer.

See Also

ent entd entl

3.5.11 entl (instruction)

Synopsis

```
entl P:P
```

Operation

```
create stack frame
```

Description

entl acts in exactly the same as *ent*. Using *entl* instead of *ent* is an optimisation hint to the VP system, indicating that the subroutine never or only rarely calls other subroutines.

Example

```
entl p0:p0
...
ret
```

Result:

```
entl has one input and one output parameter.
```

See Also

ent entd entih

3.5.12 entle (instruction)

Synopsis

```
entle P:P
```

Description

This instruction is a special form of *ente* used for a leaf subroutine. Using *entle* instead of *ente* does not change the semantics, rather is it an optimisation hint to the translator. Thus *entle* may be used for a subroutine which usually does not make any calls, but occasionally does.

3.5.13 entrytag (instruction)

Synopsis

entrytag

Description

This instruction defines a tag which must appear at the start of a prologue section to form a directly callable tool entry which is not at the start of the tool. It is not used by programmers.

3.5.14 eq (expression element)

Synopsis

```
(a eq b)
```

Operation

```
IF a equal b THEN true ELSE false
```

Description

Compares two values and returns TRUE if the first is equal to the second parameter, otherwise FALSE. *eq* can also be written as ==.

The returned value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2i*.

This operation is commutative.

Example

```
      cpy.p str1,p0
      loop
            cpy.b [p0],i1
            bcn (i1 eq 0),exitloop
            inc p0
      endloop
      printf "Should never get here\n"

exitloop:
      printf "Exit from loop. i1 = %d\n",i1
      ...
      ret

      .data

str1:
      dc.b "Example program",0
```

Result:

```
Exit from loop. i1 = 0
```

See Also

bcn bcp c2i ge geu gt gtu ne

3.6 VPcode Instructions F

3.6.1 f_i (instruction)

Synopsis

```
f_i source,destination
```

Operation

```
destination <-source BITWISE
```

Description

Moves a bit pattern from a float register to an integer register. The first parameter specifies a float register, the second an integer register.

Example

```
cpy.f 4.0,f0
f_i f0,i0   ;bit pattern moved to i0
printf "i0 = $%x\n",i0
```

Result:

```
i0 = $40800000
```

See Also

d_i i_d i_f i_l l_i

3.6.2 f2d (expression element)

Synopsis

```
(f2d a)
```

Operation

```
float to double (a)
```

Description

Converts 32 bit float value to 64 bit double.

Example

```
cpy.f 123.8956,f0
cpy (f2d f0),d0
printf "%f\n",d0
```

Result:

123.895599

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.6.3 f2ir (expression element)

Synopsis

```
(f2ir a)
```

Operation

```
Round to integer (a)
```

Description

Converts float value to integer, rounding to nearest.

Example

```
cpy.f 123.8956,f0
cpy (f2ir f0),i0
printf "%d\n",i0
```

Result:

```
124
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.6.4 f2it (expression element)

Synopsis

```
(f2it a)
```

Operation

```
truncate to integer (a)
```

Description

Converts float value to integer, truncating towards 0.

Example

```
cpy.f 123.8956,f0
cpy (f2it f0),i0
printf "%d\n",i0
```

Result:

```
123
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir i2d i2f i2l i2p l2d l2i p2i s2i

3.6.5 for (macro)

Synopsis

```
for [initialvalue],countreg
```

Operation

```
countreg = initialvalue if initialvalue defined
```

Description

for must be used in conjunction with the *next* macro. The loop count value is optional, and if it is not specified, the value already in the loop counter register *countreg* will be used as the initial count value.

The loop count is available throughout the loop in the nominated register, counting down from the initial value given to one.

Note

The exit test is performed at the end of the loop so that a loop count of zero will still execute once. If a loop count of zero is a possibility, use *while/endwhile* and adjust the loop counter manually.

The following show the code generated by the *for* macro:

for initialvalue,countreg

Source code	Generated Code
`for 25,i0`	`     cpy 25,i0`
`     ...`	`t0:`
`next i0`	`     ...`
	`     cpy (i0 sub$01),i0`
	`     bcp (i0 gt $00),t0`
	`     ...`
	`t3:`

for countreg

Source code	Generated Code
`for i0`	`t0:`
`     ...`	`     ...`
`next`	`     cpy (i0 sub$01),i0`
	`     bcp (i0 gt $00),t0`

In the second example the copy is not generated before the label. This is useful if the value is not constant but has been generated by a previous piece of code.

Example

```
;copy integer block to second location then print contents
;of new block using for/next loop

      cpy 100,i0
      qcall lib/malloc,(i0:p0)      ;new block
      cpy.p p0,p3       ;save pointer
      cpy.p intcpy,p2   ;original block
      cpy.p p2,p4       ;save pointer
      clr i2
      repeat
            cpy [p2],i1       ;get value
            add.p 4,p2  ;point to next value
            inc i2
      until.p p2>=intcpyend   ;end of table

      cpbi p4,p3,i0     ;copy block

      printf "New block now contains: "
      for i2

            cpy [p3],i1       ;get value
            add.p 4,p3  ;point to next value
            printf "%d ",i1
      next i2
      printf "\n"
      qcall lib/free,(p0:-)
      ...
      ret

      .data 4

intcpy:
      dc.i 0,0,0,-1,-1,1,1

intcpyend:
```

Result

```
New block now contains: 0 0 0 -1 -1 1 1
```

bool notbool break breakif continue continueif next if else elseif endif loop endloop repeat until while endwhile

3.7 VPCode Instructions G

3.7.1 ge (expression element)

Synopsis

```
(a ge b)
```

Operation

```
IF a GREATER THAN OR EQUAL TO b THEN true ELSE false
```

Description

Compares two signed values and returns TRUE if the first is greater than or equal to the second parameter, otherwise FALSE. *ge* can also be written as >=.

This value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2*i.

Valid types are: .i, .l, .f, .d, .p.

Pointer comparisons are unsigned.

Example

```
      cpy 10,i1
      cpy 15,i3

test1:
      bcn (i1 ge i3),exit
      inc i1
      go test1
      printf "Should never get here\n"

exit:
      printf "Test result = TRUE\n"
```

Result:

```
Test result = TRUE
```

See Also

bcn bcp c2i eq geu gt gtu le leu lt ltu ne

3.7.2 geu (expression element)

Synopsis

```
(a geu b)
```

Operation

```
IF a GREATER THAN OR EQUAL TO b THEN true ELSE false
```

Description

Compares two unsigned values and returns TRUE if the first parameter is greater than or equal to the second parameter, otherwise FALSE.

This value can be used immediately in *bcn/bcp* or it can be converted to an integer for further calculation, using *c2i*.

Valid types are: .i, .l

geu may also be used on pointers, with the same effect as *ge*.

Example

```
      cpy.l 10,l1

test1:
      bcn (l1 geu 15),exit
      inc l1
      go test1
      printf "Should never get here\n"

exit:
      printf "Test result = TRUE, so exit\n"
```

Result:

```
Test result = TRUE, so exit
```

See Also

bcn bcp c2i eq ge gt gtu le leu lt ltu ne

3.7.3 go (instruction)

Synopsis

```
go a
```

Operation

```
PC <-a
```

Description

Go to location. This is a general purpose low level unconditional jump. The parameter is either a tag in the same subroutine, or a pointer register/expression which must evaluate to a tag in the same subroutine

Example

```
      cpy 10,i1
      cpy 15,i3

test1:
      bcn (i1 ge i3),exit ;exit from loop when TRUE
      inc i1
      go test1 ;unconditional jump to 'test1'
      printf "Should never get here\n"

exit:
      printf "Exit from loop\n"
```

Result:

```
Exit from loop
```

See Also

bcn bcp

3.7.4 gos (instruction)

Synopsis

```
gos a,q:q'
```

Operation

```
subroutine register <-q
subroutine lp <-pc_return
pc <-a
pc_return:
q' <-subroutine
```

Description

Jump to subroutine. The first parameter (a) is a tag in the current tool. Like *qcall*, it takes lists of registers (q:q) to pass in, and out, of the routine. *gos* can be used to call a subroutine in the same tool. The subroutine is identified by the name of the tag just before the *ent* line.

gos can also be used with a pointer register or expression argument to call a subroutine whose address is given by the register or expression. This subroutine does not have to be in the same tool. The two typical ways of using this are to load the address of a subroutine into a pointer register directly for use by another subroutine or to use a lookup table.

Example 1

```
      ;call a subroutine in the same tool
      cpy.p str1,p4
      gos countchar,(p4:i0)
      printf "%d characters read\n",i0
      ...
      ret

countchar:
      ent p0:i0
      clr i0
      loop
            cpy.b [p0],i1
            bool ((i1 eq NUL) and (i0 ne 0)),
            exitloop
            inc p0
            inc i0
      endloop

exitloop:
      ret

      .data

str1:
      dc.b "Example program",0
```

Result:

```
15 characters read
```

Example 2

```
;call a subroutine in a different tool
tool 'tool1'
...
ent ...
...
cpy.p aroutine,p0
qcall tool2,(p0:-)
...
```

```
aroutine:
      ent i0 i1 f0 p0 : l0 d0
      ...
      ret
toolend

tool 'tool2'

      ent p0 : -
      ...
      gos p0, (i3 i5 f3 p2 : l1 d4) ;p0
      points to 'aroutine' in tool1
      ...
      ret
toolend
```

Result:

gos in tool2 calls the subroutine 'aroutine' in tool1

Example 3

```
      ;use a lookup table:
      cpy.p [(i2p (i0*4))+table],p0 ;p0 = address to call
      gos p0,(i3 i5 f3 p2:l1 d4)
      ...

routine1:
      ent i0 i1 f0 p0:l0 d0
      ...
      ret

routine2:
      ent i0 i1 f0 p0:l0 d0
      ...
      ret
      ...

.data 4
table:
      dc.p routine1
      dc.p routine2
      dc.p routine3
      ...
```

Result:

gos calls the routine at the address pointed to.

See Also

qcall

3.7.5 gose (instruction)

Synopsis

```
gose <target_expr> , <statics> , ( <input_actual_params> :
<output_actual_params> )
```

Description

This instruction is the same as *gos,* except that there is an extra operand <statics> which is an expression (default type pointer) of type pointer specifying a statics pointer value to pass to the subroutine. This is ignored by the subroutine unless it starts with *ente* or *entle*.

See also ccl, ente, gos, qcl, qcle

3.7.6 gt (expression element)

Synopsis

```
(a gt b)
```

Operation

```
IF a GREATER THAN b THEN true ELSE false
```

Description

Compares two signed values and returns TRUE if the first is greater than the second parameter, otherwise FALSE. *gt* can also be written as >.

The returned value can be used immediately in *bcn/bcp* or it can be converted to an integer for further calculation, using *c2i*.

Valid types are: .i, .l, .d, .f, .p

Pointer comparisons are unsigned.

Example

```
      clr i1
      cpy 25,i0

startloop:
      inc i1
      dec i0
      bcp (i0 gt 0),startloop

endloop:
      printf "Exit from loop; i1 = %d\n",i1
```

Result:

```
Exit from loop, i1 = 25
```

See Also

bcn bcp c2i eq ge geu gtu le leu lt ltu ne

3.7.7 gtu (expression element)

Synopsis

```
(a gtu b)
```

Operation

IF UNSIGNED a GREATER THAN UNSIGNED b THEN true ELSE false

Description

Compares two unsigned values and returns TRUE if the first is greater than the second parameter, otherwise FALSE.

The returned value can be used immediately in *bcn/bcp* or it can be converted to an integer for further calculation, using *c2i*.

gtu can be used on pointers, with the same effect as *gt* on pointers.

Example

```
      cpy.p convdata,p0
      cpy.p p0,p1
      clr i0      ;clear accumulator
      ...
      loop
            cpy ((ld.b [p0]) - '0'),i2 ;get next digit
            breakif (i2 gtu 9)      ;characters '0'..'9' are valid
            cpy ((i0 * 10) + i2),i0
            inc p0      ;point to next character
      endloop
      ...
      printf "'%s' converted to %d\n",p1,i0
      ...
      ret

      .data

convdata:
      dc.b "12.676E-6ABC",0
```

Result:

```
12.676E-6ABC converted to 12
```

See Also

bcn bcp c2i eq ge geu gt le leu lt ltu ne

3.8 VPcode Instructions I

3.8.1 i_d (instruction)

Synopsis

```
i_d source, destination
```

Operation

```
destination <-source BITWISE
```

Description

Moves bit pattern from a pair of integer registers to a double register. *source* specifies an integer register. This register (n) and the next sequential register (n+1) are concatenated and the bit pattern is moved into the double register specified by *destination*.

Example

```
cpy 1,i0
cpy 9999999,i1
i_d i0,d0   ;value in i0 and i1
printf "d0 = %g\n",d0
```

Result:

```
d0 = 8.75357e-306
```

See Also

d_i f_i i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.8.2 i_f (instruction)

Synopsis

```
i_f source,destination
```

Operation

```
destination <-source BITWISE
```

Description

Moves bit pattern from an integer register to a float register. *source* specifies an integer register and *destination* a float register.

Example

```
cpy $3f000000,i0
i_f i0,f0
printf "f0 = %g\n",f0
```

Result:

```
f0 = 0.5
```

See Also

d_i f_i i_d i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.8.3 i_l (instruction)

Synopsis

```
i_l source,destination
```

Operation

```
destination <-source BITWISE
```

Description

Moves bit pattern from a pair of integer registers to a long register. *source* specifies an integer register. This register (n) and the next sequential register (n+1) are concatenated and the bit pattern is moved into the long register specified by *destination*.

Example

```
cpy.d 4294967295.123456,d0
d_i d0,i0   ;move into i0 and i1
```

```
i_l i0,l0   ;move from i0 and i1
printf "l0 = %e\n",l0
```

Result:

```
l0 = 4.294967e+09
```

See Also

d_i f_i i_d i_f l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i s2i

3.8.4 i2d (expression element)

Synopsis

```
(i2d a)
```

Operation

```
integer_to_double
```

Description

Converts integer value to 64 bit double.

Example

```
cpy 'a',i0
cpy (i2d i0),d0
printf "d0 = %f\n",d0
```

Result:

```
d0 = 97.000000
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2f i2l i2p l2d l2i p2i s2i

3.8.5 i2f (expression element)

Synopsis

```
(i2f a)
```

Operation

```
Integer_to_float
```

Description

Converts integer value to float.

Example

```
cpy 'a',i0
cpy (i2f i0),f0
printf "f0 = %f\n",f0
```

Result:

f0 = 97.000000

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2l i2p l2d l2i p2i s2i

3.8.6 i2l (expression element)

Synopsis

```
(i2l a)
```

Operation

```
Extend to long
```

Description

Converts integer value to 64 bit long by sign extending.

Example

```
cpy 'a',i0
cpy (i2l i0),l0
printf "l0 = %ld\n",l0
```

Result:

```
l0 = 97
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2p l2d l2i p2i s2i

3.8.7 i2p (expression element)

Synopsis

```
(i2p a)
```

Operation

```
integer_to_pointer
```

Description

Converts integer value to pointer.

It should be noted that integers and pointers may have different bit patterns.

ie.

```
cpy.i, i0[mem]
cpy.p [mem], p0
```

is not equivalent to

```
cpy (i2p i0),p0
```

The exception to this is 0, which has the same representation of all 0 bits by both integers and pointers.

Example

```
cpy 'a',i0
cpy (i2p i0),p0
printf "p0 = %p\n",p0
```

Result:

```
p0 = $61
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l l2d l2i p2i s2i

3.8.8 if (macro)

Synopsis

```
if condition
```

Operation

```
if condition PC <-tag
```

Description

if must be used in conjunction with the *endif* macro. The code between these two macros is only executed when the result of the boolean expression is TRUE. For multiple conditions, the *else* and *elseif* macros are provided.

The same rules as those for *bool* apply to this macro.

The following shows the code generated by the *endif* macro:

Source code	Generated Code
`if i0=0`	`     bcp (i0 ne $00),t0`
`     ...`	`     ...`
`endif`	`t0:`

The *endif* generates the label required by the corresponding *if* and places it into the code.

Example

```
cpy '9',i0
qcall lib/isdigit,(i0:i1)
if i1 != 0
      printf "'%c' is a digit\n",i0
endif
```

Result:

```
'9' is a digit
```

See Also

else elseif endif bool notbool break breakif continue continueif for next loop endloop repeat until while endwhile

3.8.9 inc (macro)

Synopsis

```
inc operand
```

Operation

```
operand + 1
```

Description

One is added to the operand.

Valid types are: .i, .l, .f, .d, .p

Macro Expansion

```
cpy%s (%1 + 1),%1
```

Example

```
cpy 3,i1
inc i1
printf "i1=%d\n",i1
```

Result:

```
i1=4
```

See Also

dec add

3.8.10 .incbin (directive)

Includes the binary contents of a file in the output data.

Synopsis

```
.incbin "<filename>"
```

Description

This directive reads the named file, and outputs its contents as if by dc, i.e. using *litb* instructions.

See also

.align, blk, dc, litb

3.9 VPcode Instructions L

3.9.1 l_i (instruction)

Synopsis

```
l_i source,destination
```

Operation

```
destination <-source BITWISE
```

Description

Moves a bit pattern from a long register to a pair of integer registers (n,n+1). *source* specifies a long register, *destination* specifies an integer register.

Example

```
cpy.l $0010000000000000,l0
l_i l0,i0   ;value moved into i1:i0
printf "$%08x, $%08x\n",i1,i0
```

Result:

```
$00100000 $00000000
```

See Also

d_i f_i i_d i_f i_l

3.9.2 l2d (expression element)

Synopsis

```
(l2d a)
```

Operation

```
extend to double
```

Description

Convert 64 bit long value to double.

Example

```
cpy.l $0010000000000000,l0
cpy (l2d l0),d0
printf "d0=%E\n",d0
```

Result:

```
d0=4.503600E+15
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2i p2i s2i

3.9.3 l2i (expression element)

Synopsis

```
(l2i a)
```

Operation

```
truncate to integer
```

Description

Convert 64 bit long value to 32 bit integer, discarding top part.

Example

```
cpy.l $001000000000001,l0
cpy (l2i l0),i0
printf "i0=$%x\n",i0
```

Result:

```
i0 = $1
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d p2i s2i

3.9.4 ld (expression element)

Synopsis

```
(ld [effective address])
```

Operation

```
ld [effective address]
```

Description

Load from memory. Data of the specified type is loaded from the effective address.

Valid types are: .b, .d, .f, .i, .l, .p, .s, .nd, .ni, .nl, .ns

ld.b loads a 8 bit byte and zero extends into a 32 bit integer.
ld.s loads a 16 bit short and zero extends into a 32 bit integer.

Data types should match expression types with the exception of .b and .s where the expression is .i (integer).

Most data types must be naturally aligned:

- .s on a 2-byte boundary
- .f, i and .p on a 4-byte boundary
- .l and .d on an 8-byte boundary.

Four non-aligned modes are provided that do not need to be naturally aligned:

- .ns and .ni may be aligned on any byte boundary
- .nl and .nd may be aligned on any 4-byte boundary.

Example

```
      cpy.p convdata,p0
      cpy.p p0,p1
      clr i0 ;clear accumulator
      ...
      loop
            cpy ((ld.b [p0]) - '0'),i2    ;get next digit
            breakif (i2 gtu 9)      ;characters '0'..'9' are valid
            cpy ((i0 * 10) + i2),i0
            inc p0      ;point to next character
      endloop
      ...
      printf "'%s' converted to %d\n",p1,i0
      ...
      ret

      .data
convdata:
      dc.b "12.676E-6ABC",0
```

Result:

```
12.676E-6ABC converted to 12
```

See Also

cpy st

3.9.5 le (expression element)

Synopsis

```
(a le b)
```

Operation

```
IF a LESS THAN OR EQUAL TO b THEN true ELSE false
```

Description

Compares two signed values and returns TRUE if the first is less than or equal to the second parameter, otherwise FALSE.

This value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2i*.

le can also be written as <=.

Valid types are: .i, .l, .f, .d, .p.

Pointer comparisons are unsigned.

Example

```
      cpy 15,i1
      cpy 10,i3

test1:
      bcn (i1 le i3),exit
      dec i1
      go test1
      printf "Should never get here\n"

exit:
      printf "Test result = TRUE\n"
```

Result:

```
Test result = TRUE
```

See Also

bcn bcp c2i eq ge geu gt gtu leu lt ltu ne

3.9.6 leu (expression element)

Synopsis

```
(a leu b)
```

Operation

```
IF a LESS THAN OR EQUAL TO b THEN true ELSE false
```

Description

Compares two unsigned values and returns TRUE if the first parameter is less than or equal to the second parameter, otherwise it returns FALSE.

This value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2i*.

Valid types are: .i, .l

leu can also be used on pointers, with the same effect (unsigned comparison) as *le* on pointers.

Example

```
      cpy.l 10,l1

test1:
      bcn (l1 leu 15),exit
      dec l1
      go test1
      printf "Should never get here\n"

exit:
      printf "Test result = TRUE\n"
```

Result:

```
Test result = TRUE
```

See Also

bcn bcp c2i eq ge geu gt gtu le lt ltu ne

3.9.7 litb

Opcode type: instruction

```
litb
```

This instruction defines a literal block of data. It may only be used in the data section of the tool, after the first *dat* instruction. Using it in the code section of the tool results in an invalid tool.

3.9.8 liti

Opcode type: instruction

```
liti
```

This instruction defines a literal integer. It may only be used in the class hash table (which consists of at least 32 *liti* instructions after the initial *ntags* instruction at the start of the tool), or in the data section after the first *dat* instruction. Using it elsewhere results in an invalid tool.

The integer item of data produced by *liti* need not be aligned (however any *ld* expression op which loads the data is subject to the normal alignment rules). If the C is arh ARH_KNDATA (arh -2), then the tool is invalid.

3.9.9 loop (macro)

Synopsis

```
loop
```

Operation

```
None
```

Description

loop must be used in conjunction with the *endloo*p macro. This is the simplest type of loop and is most commonly used when the exit condition does not occur at the top or the bottom of the loop.

The following shows the code generated by the *loop* macro:

Source code	Generated Code

`loop`
`     ...`
`     breakif i1>'A'`
`     ...`
`endloop`
	`t0:`
	`     ...`
	`     bcn (i1 eq $41),t1`
	`     go t0:`
	`t1:`

The second label is generated because something in the loop needs to jump to it.

Example

```
      cpy.p str1,p1
      cpy 100,i0
      qcall lib/malloc,(i0:p0)
      if p0 != NULL
            cpy.p p0,p2       ;save pointer to start of buffer
            loop
                  cpy.b [p1],i1     ;get character
                  inc p1
                  continueif i1='/'       ;ignore this character
                  cpy.b i1,[p0]     ;store character
                  breakif i1=NUL
                  inc p0
            endloop
            printf "String stored = >%s<\n",p2
      endif
      qcall lib/free,(p2:-)
      ...
      ret

      .data

str1:
      dc.b "testex/break.asm",0
```

Result:

```
String stored = >testexbreak.asm<
```

See Also

endloop bool notbool break breakif continue continueif for next if else elseif endif repeat until while endwhile

3.9.10 lsl (expression element)

Synopsis

```
(a lsl b)
```

Operation

```
destination SHIFT LEFT BY source
```

Description

Shift left. The first parameter is shifted left the number of bits specified by the second parameter. Zeros are shifted in from the right hand side. The result is returned.

If the shift count equals or exceeds the data length (32 for integer, 64 for long), or is negative, then the result is undefined. A shift count of 0 is legal.

The second parameter is an integer even for the long version of the instruction.

Example

```
cpy 2,i0
cpy (i0 lsl 1), i1
printf "i1 = %d\n",i1
```

Result:

```
i1 = 4
```

See Also

asl asr asr (macro) lsl (macro) lsr lsr (macro)

3.9.11 lsl (macro)

Synopsis

```
lsl count, destination
```

Operation

```
destination <-destination SHIFT LEFT BY count
```

Description

lsl shifts the bits in the destination left by the number of places given by *count*. 0 bits are shifted in at the least significant end. Bits shifted out of the most significant end are lost.

Shift values should be in the range 0..31 for integers and 0..63 for long values; other values give undefined results.

lsl is identical to *asl.*

Valid types are: .i, .l

The first parameter is an integer.

Macro Expansion

```
cpy%s (%2 lsl %1),%2
```

Example

```
cpy 2,i0
lsl 1,i0
printf "i0 = %d\n",i0
```

Result:

```
i0 = 4
```

See Also

asr asr (macro) lsl lsr lsr (macro)

3.9.12 lsr (expression element)

Synopsis

```
(a lsr b)
```

Operation

```
a SHIFT RIGHT BY b UNSIGNED
```

Description

Shift right. The first parameter is shifted right the number of bits specified by the second parameter. Zeros are shifted in from the left hand side. The result is returned.

If the shift count equals or exceeds the data length (32 for integer, 64 for long), or is negative, then the result is undefined. A shift count of 0 is legal.

The second parameter is an integer even for the long version of the instruction.

Example

```
cpy 8,i0
cpy (i0 lsr 1), i1
printf "i1 = %d\n",i1
```

Result:

```
i1 = 4
```

See Also

asl asr asr (macro) lsl lsl (macro) lsr (macro)

3.9.13 lsr (macro)

Synopsis

```
lsr count,destination
```

Operation

```
destination <-destination UNSIGNED SHIFT RIGHT BY count
```

Description

lsr shifts the bits in *destination* right by the number of places given by *count*. Bits shifted out of the least significant end are lost, bits shifted in to the most significant end are set to 0.

The first parameter is an integer.

Shift values should be in the range 0..31 for integers and 0..63 for long values; other values give undefined results.

Valid types are: .i, .l

lsr is identical to *asr*

Macro Expansion

```
cpy%s (%2 lsr %1),%2
```

Example

```
cpy 8,i0
lsr 1,i0
printf "i0 = %d\n",i0
```

Result:

```
i0 = 4
```

See Also

asl asr asr (macro) lsl lsl (macro) lsr

3.9.14 lt (expression element)

Synopsis

```
(a lt b)
```

Operation

```
IF a LESS THAN b THEN true ELSE false
```

Description

Compares two signed values and returns TRUE if the first is less than the second parameter, otherwise it returns FALSE.

The returned value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2i.*

lt can also be written as <.

Valid types are: .i, .l, .d, .f, .p

Pointer comparisons are unsigned.

Example

```
      clr i0

startloop:
      inc i0
      bcp (i0 lt 25),startloop

endloop:
      printf "Exit from loop; i0 = %d\n",i0
```

Result:

```
Exit from loop, i0 = 25
```

See Also

bcn bcp c2i eq ge geu gt gtu le leu ltu ne

3.9.15 ltu (expression element)

Synopsis

```
(a ltu b)
```

Operation

```
IF UNSIGNED a LESS THAN UNSIGNED b THEN true ELSE false
```

Description

Compares two unsigned values and returns TRUE if the first is less than the second parameter, otherwise it returns FALSE.

The returned value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2i*.

ltu can be used on pointers with the same effect as *lt* on pointers.

Example

```
clr i1
while (i1 ltu 10)
      inc i1
endwhile
printf "i1=%d\n",i1
```

Result:

```
i1=10
```

See Also

bcn bcp c2i eq ge geu gt gtu le leu lt ne

3.10 VPcode Instructions M

3.10.1 mul (expression element)

Synopsis

```
(a mul b)
```

Operation

```
return a * b
```

Description

Multiplies the two parameters and returns the result.

This operation is commutative.

The assembler allows the use of * instead of *mul*.

Example

```
cpy.d 10.5,d1
cpy (d1 mul 3),d1
printf "d1=%f\n",d1
```

Result:

```
d1 = 31.500000
```

See Also

add add (macro) div div (macro) divh divu mul (macro) mulh sub sub (macro)

3.10.2 mul (macro)

Synopsis

```
mul source,destination
```

Operation

```
destination <-destination * source
```

Description

mul performs a signed multiplication of *destination* and *source*, leaving the result in *destination*.

Valid types are: .i, .l, .f, .d

Macro Expansion

```
cpy%s (%2 mul %1), %2
```

Example

```
cpy.d 10.5,d1
mul.d 3.0,d1
printf "d1=%f\n",d1
```

Result:

```
d1=31.500000
```

See Also

add add (macro) div div (macro) divh divu mul mulh sub sub (macro)

3.10.3 mulh (expression element)

Synopsis

```
(a mulh b)
```

Operation

```
(a * b)/65536
(a * b)>>16 (signed)
```

Description

Multiplies the two parameters using fixed point arithmetic. The effect is to multiply the parameters using integer arithmetic with a 48 bit result, and then shift the result 16 bits right.

For positive results, any bits shifted out of the result register are ignored; rounding is towards 0.

For negative results, rounding will be either downwards or towards 0. This behaviour is undefined, but worse case error is less than 1 in the least significant bit.

This operation is commutative.

Example

```
cpy (27 * 65536),i1
cpy (i1 mulh 3),i1
printf "i1=%d\n",i1
```

Result:

```
i1 = 81
```

See Also

add add (macro) div div (macro) divh divu mul mul (macro) sub sub (macro)

3.11 VPcode Instructions N

3.11.1 nbit (expression element)

Synopsis

```
(a nbit b)
```

Operation

```
IF (a AND (1 << )) = 0 THEN true ELSE false
```

Description

Test bit. The bit number specified by the second parameter, *b*, is examined in the first parameter, *a*. If this is set, FALSE is returned. If not set, TRUE is returned.

b can be a constant, register or expression. The result is undefined if *b* is outside the range 0..31

The returned value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2i*.

Example

```
      cpy $7f800000,i1
      bcn (i1 nbit 16),bitset
      printf "Bit 16 is set\n"
      go exit

bitset:
      printf "Bit 16 is not set\n"

exit:
```

Result:

```
Bit 16 is set
```

See Also

bit c2i bcn bcp

3.11.2 ncall (macro)

Synopsis

```
ncall pointer, name, parameter list
```

Description

ncall is used to invoke a method on an object (A method defines an operation performed by an object of a particular class. Methods may be accessible to sub-classes and/or other classes. A sub-class may override (replace) a method of its base or parent class).

By convention, the object pointer is also passed as the first parameter to the method. The *ncall* macro finds the class tool for the object (in [*p3+ob_class*]) and attempts to invoke the named method in that class. If the class does not have such a method, execution falls through to the class's default method, which typically uses the *parentclass* macro to attempt to invoke the same method in the parent class, and so on until the method is found.

Example

```
ncall p3, newmethod, (p3 p7 i2:i3)
```

Result:

Invokes the method 'newmethod' on the object p3.

See Also

ccall parentclass pcall qcall

3.11.3 ne (expression element)

Synopsis

```
(a ne b)
```

Operation

```
IF a not equal b THEN true ELSE false
```

Description

Compares two values and returns TRUE if the first is not equal to the second parameter, otherwise FALSE.

The returned value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2i*.

ne can also be written as !=.

This operation is commutative.

Example

```
      cpy.p str1,p0
      clr i0
      loop
            cpy.b [p0],i1
            bool ((i1 eq NUL) and (i0 ne 0)),
      exitloop
      inc p0
      inc i0

      endloop

exitloop:
      printf "%d characters read\n",i0
      ...
      ret

str1:
      dc.b "Example program",0
```

Result:

```
15 characters read
```

See Also

c2i bcn bcp eq ge geu gt gtu

3.11.4 neg (macro)

Synopsis

```
neg operand
```

Operation

```
operand <-operand
```

Description

neg subtracts the operand from 0 and returns the result to the operand.

For integers and longs, this forms the two's complement of the number.

Valid types are: .i, .l, .f, .d

Macro Expansion

```
cpy%s (%1*(-1)),%1
```

Example

```
cpy -1,i0
neg i0 ;make +ve
printf "i0 = %d\n",i0
```

Result:

```
i0 = 1
```

See Also

not

3.11.5 next (macro)

Synopsis

```
next %1
```

Operation

```
dec %1 PC <-tag of for if %1 > 0
```

Description

next is used in conjunction with the *for* macro. The loop count register must be the same as specified by the corresponding for macro.

The following shows the code generated by the *for* macro:

for *constant,register*

Source code	Generated Code

`for 25,i0`
`     ...`
`next i0`
	`     cpy $19,i0`
	`t0:`
	`     ...`
	`          cpy (i0 sub$01),i0`
	`          bcp (i0 gt$00),t0`
	`     ...`
	`t3:`

for *register*

Source code	Generated Code
for i0 ... next	t0: ... cpy (i0 sub$01),i0 bcp (i0 gt $00),t0

In the second example the copy is not generated before the label. This is useful if the value is not constant but has been generated by a previous piece of code.

Example 1

```
;copy integer block to second location then print contents
;of new block using for/next loop

      cpy 100,i0
      qcall lib/malloc,(i0:p0)      ;new block
      cpy.p p0,p3       ;save pointer
      cpy.p intcpy,p2   ;original block
      cpy.p p2,p4       ;save pointer
      clr i2
      repeat
            cpy [p2],i1 ;get value
            add.p 4,p2 ;point to next value
            inc i2
      until.p p2>=intcpyend ;end of table

      cpbi p4,p3,i0 ;copy block

      printf "New block now contains: "
      for i2
            cpy [p3],i1 ;get value
            add.p 4,p3 ;point to next value
            printf "%d ",i1
      next i2
      printf "\n"
      qcall lib/free,(p0:-)
      ...
      ret

      .data 4

intcpy:
      dc.i 0,0,0,-1,-1,1,1

intcpyend:
```

Result

```
New block now contains: 0 0 0 -1 -1 1 1
```

See Also

for bool notbool break breakif continue continueif if else elseif endif loop endloop repeat until while endwhile

3.11.6 noret (instruction)

```
noret
```

This instruction marks a point in the code where it is known execution can never be reached. It is typically used after a *gos, gose, qcl, qcle* or *ccl* which is known never to return. If execution does reach a *noret* instruction, undefined behaviour ensues.

Omitting this instruction after a call which is known not to return does not alter the semantics of the code, but may make the translated code less optimised.

3.11.7 not (macro)

Synopsis

```
not operand
```

Operation

```
operand <-BITWISE NOT OF operand
```

Description

The ones complement of the operand is placed in the operand.

Valid types are: .i, .l

Macro Expansion

```
cpy%s (%1 xor -1),%1
```

Example

```
cpy 8,i2
not i2
printf "i2 = %d\n",i2
```

Result:

```
i2 = -9
```

See Also

neg not xor

3.11.8 notbool (macro)

Synopsis

```
notbool condition,label
```

Operation

```
if condition FALSE then PC &- label
```

Description

notbool performs the same function as *bool*, except that it makes the jump if the boolean expression returns FALSE. It is the same as *bc not(..)* – the assembler handles it by inverting the condition being negated.

Example

```
      cpy.p str1,p0
      loop
            cpy.b [p0],i1
            notbool i1 != NUL, exit
            inc p0
      endloop
      ...
```

```
exit:
      printf "Exit from loop, i1 = %d\n",i1
      ret

      .data

str1:
      dc.b "Example program",0
```

Result:

```
Exit from loop, i1 = 0
```

See Also

bool break breakif continue continueif for next if else elseif endif loop endloop repeat until while endwhile

3.11.9 ntags (instruction)

Synopsis

```
ntags C
```

Operation

None

Description

Specifies the number of tags needed by this tool. This should be the first instruction in the code.

Note

This instruction is not used by the programmer, the assembler will automatically add this in the *tool* macro.

See Also

tag

3.12 VPcode Instructions O

3.12.1 or (expression element)

Synopsis

```
(a or b)
```

Operation

```
a BITWISE OR b
```

Description

Bitwise *or*. The two parameters are bitwise *or*ed and the result is returned.

Bitwise *or* may also be written as |.

This operation is commutative.

Example

```
clr i2
cpy (i2 or 8),i2
printf "i2 = %d\n",i2
```

Result:

```
i2 = 8
```

See Also

or (macro) or c xor xor (macro)

3.12.2 or (macro)

Synopsis

```
or source,destination
```

Operation

```
destination <-destination BITWISE OR source
```

Description

or performs the logical inclusive or of the two operands and leaves the result in destination. A bit in the result is set if either of the corresponding bits in the original operands is set; otherwise the result bit is cleared.

Valid types are: .i, .l

Macro Expansion

```
cpy%s (%2 or %1),%2
```

Example

```
clr i2
or 8,i2
printf "i2 = %d\n",i2
```

Result:

```
i2 = 8
```

See Also

or or c xor xor (macro)

3.12.3 or conditional (expres sion element)

Synopsis

```
(a or b)
```

Operation

```
a BOOLEAN OR b
```

Description

Condition (boolean) *or*. The two parameters are logically *or*ed and the result is returned.

Both parameters may be evaluated before this completes. Certain translators may evaluate only one parameter when the result is clear from that one parameter. If it is desired to avoid executing some code (e.g. a load from memory) if one condition is true, a *bc* instruction should be used.

As in *and* conditional, C shortcut rules apply – in “A or B”, if A is true B is never evaluated.

The conditional *or* may also be written as **||**.

This operation is commutative.

Example

```
cpy 21,i0
if ((i0 gt 31) or (i0 lt 1))
      printf "Range error\n"
else
      printf "No error\n"
endif
```

Result:

```
No error
```

See Also

or or (macro) xor xor (macro)

3.12.4 ord (expression element)

Synopsis

```
(ord a)
```

Operation

```
IF ordered (a) THEN true ELSE false
```

Description

Ordered. Return TRUE if the value is a valid number i.e. in the range -infinity...+infinity (+INF...-INF), and FALSE if it is Not-a-Number (NaN).

The returned value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2i*.

Valid types are : .f, .d

Example

```
      cpy.d -8.0e300,d0       ;ordinary number
      bcn (ord d0),numexit
      printf "Not-a-Number\n"
      go exit
      ...

numexit:
      printf "Number is in range +INF..-INF\n"

exit:
```

Result:

```
Number is in range +INF..-INF
```

See Also

c2i bcn bcp uno

3.13 VPcode Instructions P

3.13.1 p2i (expression element)

Synopsis

```
(p2i a)
```

Operation

```
to_integer(a)
```

Description

Convert pointer to integer.

It should be noted that integers and pointers may have different bit patterns. The exception to this is 0, which does have the same representation of all 0 bits in both integers and pointers.

Example

```
      cpy.p str1,p0
      cpy.p p0,p1       ;p1 points to beginning of string
      cpsb p0,p0  ;point to NUL terminator
      cpy ((p2i p0)-(p2i p1)),i0
      tracef "Source: [%s], Length %d\n",p1,i0
      ...
      ret

      .data

str1:
      dc.b " 8967ASDkfjdRT b363 ",0
```

Result:

```
Source: [ 8967ASDkfjdRT b363 ], Length 22
```

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i s2i

3.13.2 parentclass (macro)

Synopsis

```
parentclass
```

Description

parentclass is used in a default method (an *entd* block) to chain to the same method in the parent class. The typical code for a default method in any class that inherits from another class (i.e. it has a parent) is:

```
entd
parentclass
```

```
ret
```

See Also

ccall ncall pcall qcall

3.13.3 pcall (macro)

Synopsis

```
pcall pointer, methodname, registers
```

Description

pcall is used within a class tool to invoke a method in the parent class.

Example

```
method nameindex
ent p0 p1 i0 : i0
pcall nameindex, (p0 p1 i0 : i0)    ;invoke parent's nameindex method
ret
```

See Also

ccall ncall parentclass qcall tracef

3.13.4 pri (instruction)

Synopsis

```
pri a
```

Operation

```
pri(a)
```

Description

Private. Access to operating system feature, or low level machine specific feature. This is not a portable operation, but would normally be accessed via library calls.

Note

This instruction is not used by the programmer.

3.14 VPcode Instructions Q

3.14.1 qcall (macro)

Synopsis

```
qcall tool,(q:q) [,flags]
```

Operation

```
Quick call.
```

Description

The *qcall* macro takes the name of a tool as a parameter, followed by the input and output registers. The programmer specifies appropriate registers for each individual application or routine.

```
qcall lib/argcargv,(-:p0 i0)
```

The tool is loaded and bound when the application referencing it is loaded. It will remain available in local memory for at least as long as the referencing application is in memory.

qcall has an optional third parameter that affects how the called tool is loaded:

• No third parameter	Normal call. Called tool is loaded at the same time as the calling tool.
• VIRTUAL	Called tool is searched for and possibly loaded each time the call is made. Much slower, but good for an infrequently called large tool since the called tool does not take any memory until it is needed, and the kernel can free it again afterwards if memory is low. This technique is often used in programs that have a number of subroutines, only one of which will be executing at a time.
• VIRTUAL+FIXUP	Called tool is searched for and possibly loaded the first time this call is executed, but is then kept in memory. Slow on the first call but nearly as fast as normal calls thereafter. The advantage of this method is that the called tool's code is not loaded into memory, translated and bound until it is actually called. This can significantly speed up the initial startup of a program that references many tools. The other common reason for using this method is in the case where a tool references a number of other tools, but only one of them will be needed in a particular system, due to a set of constraints (such as, for example, screen pixelmaps on a particular system - it would not be necessary to load screen pixelmaps for a PC on an embedded system, since they would never be called). By using virtual+fixup calls, the tools that are not used are never loaded.

If the *flags* argument is not specified, the default value is 0, or a normal call. However this can be globally overridden by setting *__qcall_default_flags* to a different default value.

Example 1

```
qcall app/mydirectory/tool2,(p0:-),VIRTUAL
```

Result:

'tool2' is possibly loaded, translated and bound each time it is called.

Example 2

```
qcall app/mydirectory/tool2,(p0:-),VIRTUAL+FIXUP
```

Result:

'tool2' is loaded and kept in memory.

See Also

ent entd ccall ncall parentclass pcall qcalle qcallv qcl ret

3.14.2 qcalle (macro)

Synopsis

```
qcall tool, statics, (q:q) [, flags]
```

Description

The *qcalle* macro uses the *qcle* instruction to call another tool.

The *statics* expression (default type pointer, result must be type pointer) specifies a statics pointer to pass into the called tool.

The actual parameters are specified in the same way as in the *gos* instruction.

The optional flags argument is specified in the same way as in the *qcall* macro.

See Also

ent entd ccall ncall parentclass pcall qcall qcallv qcle ret

3.14.3 qcallv (macro)

Synopsis

```
qcallv tool, (q:q) [ , varargs]
```

Description

This macro puts the *varargs* (currently) onto the stack, then uses *qcall* to call the specified tool.

See Also

ent entd ccall ncall parentclass pcall qcall qcalle ret

3.14.4 qcl (instruction)

Synopsis

```
qcl CQQ
```

Operation

```
Quick call.
```

Description

Used by macro *qcall.*

Note

This instruction is not used by the programmer; the assembler macro is *qcall*
See Also

call

3.14.5 qcle (instruction)

Synopsis

```
qcle CRQQ CEQQ
```

Description

This instruction calls another tool. It works like *qcl*, but there is an extra actual input pointer parameter (the extra second operand in the encoding), which can be a register or an expression. If the target subroutine (possibly via any intervening *entd* subroutines) is an *ente* or *entle* subroutine, the extra pointer becomes the statics pointer in the subroutine. Otherwise, the extra parameter is ignored.

3.15 VPcode Instructions R

3.15.1 rem (expression element)

Synopsis

```
(a rem b)
```

Operation

```
remainder(a/b) SIGNED
```

Description

Returns remainder. The first parameter is divided by the second parameter using signed arithmetic and the remainder is returned.

(a rem b) is identical to *(a-(b*(a div b)))*

Dividing by zero gives the processor exception EXC_INT_DIV_0, which becomes a SIGFPE signal if uncaught.

Valid types are: .i, .l

Example

```
cpy 13,i0
cpy (i0 rem 6),i1
printf "%d rem 6=%d\n",i0,i1
```

Result:

```
13 rem 6 = 1
```

See Also

div div (macro) divh divu rem (macro) remu remu (macro)

3.15.2 rem (macro)

Synopsis

rem source,destination

Operation

```
remainder(destination/source)SIGNED
```

Description

Remainder. *source* is divided by *destination* using signed arithmetic and the remainder is placed in destination.

Dividing by zero gives the processor exception EXC_INT_DIV_0, which becomes a SIGFPE signal if uncaught.

Macro Expansion

```
cpy%s (%2 rem %1),%2
```

Example

```
cpy 13,i0
rem 6,i0    ;remainder in i0
printf "i0=%d\n",i0
```

Result:

```
i0 = 1
```

See Also

div div (macro) divh divu rem remu remu (macro)

3.15.3 remu (expression element)

Synopsis

```
(a remu b)
```

Operation

```
remainder(a/b) UNSIGNED
```

Description

Unsigned remainder. The first parameter is divided by the second parameter using unsigned arithmetic and the remainder is returned.

(a remu b) is identical to *(a-(b*(a divu b)))*

Dividing by zero gives the processor exception EXC_INT_DIV_0, which becomes a SIGFPE signal if uncaught.

Example

```
cpy 13,i0
cpy (i0 remu 6),i1
printf "%d remu 6=%d\n",i0,i1
```

Result:

```
13 remu 6 = 1
```

See Also

div div (macro) divh divu rem remu (macro)

3.15.4 remu (macro)

Synopsis

```
remu source,destination
```

Operation

```
remainder(destination/source)UNSIGNED
```

Description

Remainder. The first parameter is divided by the second parameter using unsigned arithmetic and the remainder is returned.

Dividing by zero gives the processor exception EXC_INT_DIV_0, which becomes a SIGFPE signal if uncaught.

Macro Expansion

```
cpy%s (%2 rem %1), %2
```

Example

```
cpy 13,i0
remu 6,i0
printf "i0=%d\n",i0
```

Result:

```
i0 = 1
```

See Also

div div (macro) divh divu rem remu

3.15.5 repeat (macro)

Synopsis

```
repeat
```

Operation

None

Description

repeat must be used in conjunction with the *until* macro. The exit condition is tested at the bottom of the loop by the *until* macro.

The block is executed one or more times.

The following shows the code generated by the *repeat* macro:

Source code	Generated Code

`repeat`
`     ...`
`until`
	`t0:`
	`     ...`
	`     bcp (...),`
	`t0`

Example

```
      clr i0
      cpy.p str1,p1
      repeat
            cpy.b [p1],i1     ;get char
            inc p1      ;advance
            inc i0
      until i1=NUL
      printf "i0 = %d\n",i0
      ret

      .data
str1:
      dc.b "abcefABCD",0
```

Result:

```
i0=10
```

See Also

until bool notbool break breakif continue continueif for next if else elseif endif loop endloop while endwhile

3.15.6 ret (instruction)

Synopsis

```
ret
```

Operation

```
pc <-lp undo stack frame created by ent/aframe/als
```

Description

Return from routine. The stack frame (as created by *ent, als* etc.) is removed and control is returned to the caller. Registers are returned as specified by *ent* and also by the calling instruction.

Example

```
ent p0 i0-i2:i0
...
...
ret
```

Result:

Control is returned to caller.

See Also

ent entd entl als qcall

3.16 VPcode Instructions S

3.16.1 s2i (expression element)

Synopsis

```
(s2i a)
```

Operation

```
extend to integer
```

Description

Converts short value to integer by sign extending.

Note

If it is desired to 0 extend a short, an *and* may be used.

Example 1

```
cpy (s2i i0),i1
```

Result:

Sign extends the bottom 16 bits of i0 into i1.

The most common use is when loading a short.

Example 2

```
cpy (s2i(ld.s[p0])),i1
```

Loads a short from [p0], sign extends it to an integer and puts the answer in i1.

See Also

d_i f_i i_d i_f i_l l_i b2i d2f d2ir d2it d2lr d2lt f2d f2ir f2it i2d i2f i2l i2p l2d l2i p2i

3.16.2 saveall (private instru ction)

Synopsis

```
saveall
```

This instruction has a specialist function within the Elate kernel's scheduler.

Normally, a native tool (a tool written in native assembler) must conform to the calling convention for the processor's implementation of Elate, including preserving certain processor registers.

However it is possible to call a native tool that trashes some of the registers that the calling convention dictates that it must preserve. To do this, the calling subroutine must contain a saveall instruction. On return from the native tool to the calling subroutine, all VP registers except sp, lp, pp and gp are trashed.

This is a private instruction, meaning that it is not expected to be used outside the Elate kernel.

3.16.3 schk (instruction)

Synopsis

```
schk constant (size)
```

Operation

```
check stack size
```

Description

Stack check and reallocate.

schk takes a constant parameter - the maximum number of bytes of stack that is needed when calling out from the routine to another routine.

The *schk* instruction must be the first instruction after the entry instruction as it affects the whole of the *ent* block.

Register *pp* may be corrupted by this instruction.

Example

```
ent p0 i0 p1:i0
```

```
schk 100
...
qcall tool1,(p0:i1)
...
ret
```

Result:

schk checks there is enough stack for this subroutine, plus an extra 100 bytes to allow for tool1's stack usage.

See Also

None

3.16.4 st (expression element)

Synopsis

```
(st [effective address])
```

Operation

```
st data, [effective address]
```

Description

Store to memory. Data is written to the effective address.

Most data types must be naturally aligned:

- .s on a 2-byte boundary
- .f, i and .p on a 4-byte boundary
- .l and .d on an 8-byte boundary

Four non-aligned modes are provided that do not need to be naturally aligned:

- .ns and .ni may be aligned on any byte boundary
- .nl and .nd may be aligned on any 4-byte boundary

Note

This instruction is not used by the programmer, as it is generated by the assembler when using *cpy* to a memory location.

See Also

cpy ld

3.16.5 sub (expression element)

Synopsis

```
(a sub b)
```

Operation

```
return a - b
```

Description

Subtract. The second parameter is subtracted from the first and the result is returned.

It is possible to calculate both pointer difference (pointer *sub* pointer returning an integer) and pointer subtraction (pointer *sub* integer returning pointer).

The assembler allows the use of the symbol "-" instead of *sub.*

Example

```
cpy 10,i0
cpy (i0 sub 6),i1
printf "%d sub 6 = %d\n",i0,i1
```

Result:

```
10 sub 6 = 4
```

See Also

add add (macro) div div (macro) divh divu mul mul (macro) mulh sub (macro)

3.16.6 sub (macro)

Synopsis

```
sub source,destination
```

Operation

```
destination <-source - destination
```

Description

Subtract. *destination* is subtracted from *source* leaving the result in the *destination*.

Macro Expansion

```
cpy%s (%2 sub %1), %2
```

Example

```
cpy 10,i0
sub 6,i0
printf "i0 = %d\n",i0
```

Result:

```
i0 = 4
```

See Also

add add (macro) div div (macro) divh divu mul mul (macro) mulh sub

3.16.7 swb (expression element)

Synopsis

```
(swb i)
```

Operation

```
swap bytes
```

Description

Swap bytes within integer. Converts from big endian to little endian or vice versa.

Example

```
cpy $123456,i0
cpy (swb i0),i0
printf "$%x\n",i0
```

Result:

```
$56341200
```

3.16.8 sync (instruction)

Synopsis

```
sync
```

Operation

```
Synchronise
```

Description

This instruction has two uses:

- It must be used immediately after the target of a non local *goto*, either after a call to *lib/longjmp* or after an exception catch tag.
- In the data section of a tool, it causes an align to a four byte boundary. The VP programmer does not use *sync* directly in this way, since the assembler's *.align* directive does the same thing.

Example

```
ent p0 i0 : -
als JMPBUF_SIZE ;allocate stack
...
qcall lib/setjmp,sp:i1
sync
...
syncreg p0-p1 i0
...
qcall lib/longjmp ;(or call something that does this)
...
syncreg -
...
ret
```

Result:

In this example, a call to *lib/longjmp* will end up back just after the *setjmp*.

See Also

data syncreg

3.16.9 syncreg (instruction)

Synopsis

```
syncreg Q
```

Description

syncreg marks which VP registers should not be trashed when a non local *goto* is taken.

Each instruction in a subroutine has a *syncreg state*. The *syncreg* state of each instruction is determined by lexically analysing the subroutine (as opposed to following program flow) as follows:

1. Before the first *syncreg* instruction in the subroutine, all instructions have an empty syncreg state.
2. Each *syncreg* instruction causes all following instructions up to the next *syncreg* instruction (or the end of the subroutine if that occurs first) to have a *syncreg* state consisting of the list of registers in the Q operand.
3. The *syncreg* instruction allows a special form of the Q operand, a single byte $08, to mean "all local registers."

Note

All VP registers (except the one being returned by *setjmp*) are trashed by a non-local jump. Failure to use this instruction could corrupt registers.

Example

```
ent p0 i0 : -
als JMPBUF_SIZE   ;allocate stack
...
qcall lib/setjmp,sp:i1
sync
...
syncreg p0-p1 i0
...
qcall lib/longjmp ;(or call something that does  this)
...
syncreg -
...
ret
```

Result:

The *syncreg*s are used to say which registers should be preserved when the *longjmp* happens - by default.

See Also

data sync

3.17 VPcode Instructions T-Z

3.17.1 tag (instruction)

Synopsis

```
tag
```

Operation

None

Description

Defines a tag (label) at the current location. The tag number defined is the next sequential number, the first tag being number 0.

Note

This instruction is not used by the programmer - it is generated by defining a label.

3.17.2 toolentry (instruction)

Marks a directly callable tool entry.

Synopsis

```
toolentry
```

Description

This "instruction" forces the next tag to be emitted as a entrytag instruction. This has the effect of making the tag a directly callable tool entry, which means that fixups in other tools are allowed to refer to this tag, assuming that some mechanism has been set up to allow the fixup mechanism to find the tag.

In the VP specification, such a tag may only appear at the start of a prologue.

See also

entrytag.

3.17.3 uno (expression element)

Synopsis

```
(uno a)
```

Operation

```
IF unordered(a) THEN true ELSE false
```

Description

Unordered. Return TRUE if the value is a Not-a-number (NaN), and FALSE otherwise i.e. if it is in the range -infinity..+infinity (+INF...-INF).

The returned value can be used immediately in *bcn*/*bcp* or it can be converted to an integer for further calculation, using *c2i.*

Valid types are: .d, .f

Example

```
      cpy.d -8.0e300,d0 ;ordinary number
      bcn (uno d0),numexit
      printf "Number is in range +INF..-INF\n"
      go exit
      ...

numexit:
      printf "Not-a-Number\n"

exit:
```

Result:

```
Number is in range +INF..-INF
```

See Also

c2i bcn bcp ord

3.17.4 until (macro)

Synopsis

```
until condition
```

Operation

```
if not condition PC <-tag of repeat
```

Description

until must be used in conjunction with the *repeat* macro. The loop is terminated when the result of the boolean expression is TRUE.

The same rules as those for *bool* apply to this macro.

The following shows the code generated by the *until* macro:

Source code	Generated Code

repeat
    ...
until
	t0:
	     ...
	     bcp (...),t0

Example

```
clr i0

      cpy.p str1,p1
      repeat
            cpy.b [p1],i1     ;get char
            inc p1      ;advance
            inc i0
      until i1=NUL
      tracef "i0 = %d\n",i0
      ret

      .data

str1:
      dc.b "abcefABCD",0
```

Result:

```
i0=10
```

See Also

while bool notbool break breakif continue continueif for next if else elseif endif loop endloop repeat endwhile

3.17.5 while (macro)

Synopsis

```
while condition
```

Operation

```
if not condition PC <-tag of endwhile
```

Description

while must be used in conjunction with the *endwhile* macro. The loop is only executed while the boolean expression is true. The check is done at the top of the loop so the block may be executed zero or more times.

The same rules as those for *bool* apply to this macro.

The following shows the code generated by the *while* macro:

Source code	Generated Code

`while i1!=NUL`
`     ...`
`endwhile`
	`t0:`
	`     bcn (i1 eq $00),t1`
	`     ...go t0`
	`t1:`

Example

```
      cpy.p intcpy,p0   ;start of table
      cpy.p intcpyend,p1      ;end of table
      while p0 != p1     ;exit when p0=p1
            cpy [p0],i1       ;get value
            add.p 4,p0
            printf "%d ",i1
      endwhile    ;exit when p0=p1
      printf "\n"
      ...
      ret

      .data 4

intcpy:
      dc.i 0,0,0,-1,-1,1,1

intcpyend:
```

Result:

```
0 0 0 -1 -1 1 1
```

See Also

endwhile bool notbool break breakif continue continueif for next if else elseif endif loop endloop repeat until endwhile

3.17.6 xor (expression eleme nt)

Synopsis

```
(a xor b)
```

Description

Bitwise exclusive *or*. The two parameters are bitwise exclusive *or*ed together and the result is returned.

xor can also be written as ^.

This operation is commutative.

Example

```
clr i0
cpy (i0 xor 8),i0
printf "i0 = %d\n",i0
```

Result:

```
i0 = 8
```

See Also

not (macro) or or (macro) xor (macro)

3.17.7 xor (macro)

Synopsis

```
xor source,destination
```

Operation

```
destination <-destination BITWISE OR source
```

Description

xor performs the logical exclusive or of the two operands and returns the result in *destination*.

This operation is commutative.

Macro Expansion

```
cpy%s (%2 xor %1),%2
```

Example

```
clr i0
xor 8,i0
printf "i0 = %d\n",i0
```

Result:

```
i0 = 8
```

See Also

not (macro) or or (macro)

3.17.8 zap (instruction)

Synopsis

```
zap a
```

Operation

```
a <-?
```

Description

This instruction "zaps" its operand register, i.e. marks that the value in the register is dead and will not be used.

A register can be "zapped" by marking its last use with a '~'. However there are occasions where this is not possible, for example if the last use of a value is in an instruction where the value is used twice, or the last use is in the last iteration of a loop. In these cases a separate *zap* instruction must be used.

Zapping a register causes its value to become undefined. Reading an undefined register generates an undefined value, possibly different each time it is done.

Example

```
zap i2
```

Result:

Indicates that the value in *i2* is no longer needed.

See Also

None